A C TO RTL ALGORITHM USING STRUCTURED CIRCUIT TEMPLATES: A CASE STUDY WITH SIMULATED ANNEALING

by

Jonathan D. Phillips

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Dr. Aravind Dasu Major Professor	Dr. Brandon Eames Committee Member
Dr. Nicholas Flann Committee Member	Dr. Stephen Allan Committee Member
Dr. Charles Swenson Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2008

Copyright © Jonathan D. Phillips 2008

All Rights Reserved

Abstract

A C to RTL Algorithm Using Structured Circuit Templates: A Case Study with Simulated Annealing

by

Jonathan D. Phillips, Doctor of Philosophy

Utah State University, 2008

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

A tool flow is presented for deriving accelerator circuits on an FPGA from ANSI C source code by exploring architecture solutions that conform to a preset template through scheduling and mapping algorithms. A case study carried out on simulated annealing-based AMPS software used for spacecraft systems is explained. The goal of the tool is the derivation of a design that maximizes throughput while minimizing footprint. Results obtained are compared with a peer C to RTL tool, a space-borne embedded processor and a commodity desktop processor for a variety of problems.

(127 pages)

Contents

Page

Abstract . . . . . **iii**

List of Tables . . . . . **vi**

List of Figures . . . . . **vii**

1 Introduction **1**
1.1 Motivation 1
1.2 Principal Contributions 2
1.3 Overview 2

2 Background **3**
2.1 FPGAs 3
2.2 C-to-Hardware Compilers 4
2.3 High Level Synthesis Techniques 9
2.4 Iterative Repair and Simulated Annealing 13
2.5 FPGAs in the Space Environment 23

3 Identifying a Hardware Template **25**
3.1 The Memory Sub-System, Data-Routing Module, and Main Controller . . . 26
3.2 The Copy Stage 29
3.3 The Alter Stage 31
3.4 The Evaluate Stage 33
3.5 The Accept Stage and the Adjust Temperature Module 37
3.6 Summary of the Architecture and System Performance Characterization . . 39
3.7 Differences between Software and Hardware Implementations 42
3.8 Enhancements through PDR and TMR 42
3.8.1 PDR applied to the Evaluate sub-system 42
3.8.2 TMR applied to the entire AMPS accelerator circuit 43
3.9 Performance of the AMPS Accelerator Circuit 45

4 Architecture Derivation Methodology **51**
4.1 Functional Block Partitioner 53
4.2 Constant Extractor 53
4.3 GCC 53
4.4 CDFG Generator and Optimizer 55
4.5 Area and Timing Metadata 57
4.5.1 Estimating Resource Usage 57
4.5.2 Determining Relative Resource Costs 70
4.6 Scheduling and Mapping (SAM) 72

4.6.1 Introduction . . . 72
4.6.2 Possible SAM Algorithms . . . 75
4.6.3 SAM Method of Choice . . . 79
4.7 HIF to VHDL Converter . . . 83

5 Results . . . 84
5.1 Test Cases . . . 84
5.1.1 Traveling Salesperson . . . 84
5.1.2 Graph Coloring Problem . . . 85
5.1.3 Dependency Graph Violation Removal Problem . . . 87
5.2 SATH vs. Impulse/PPC/X86 . . . 88

6 Conclusions and Future Work . . . 95

References . . . 96

Appendices . . . 103
Appendix A Iterative Repair C Code . . . 104
Appendix B TSP C Code . . . 109
Appendix C Graph Coloring C Code . . . 112
Appendix D Dependency Graph Violation C Code . . . 115

List of Tables

Table		Page
2.1	Truth Table for Eq. (2.1)	4
3.1	Variable Definitions	26
3.2	Reconfiguration Time of Evaluate and AMPS Accelerator Circuit	48
3.3	Architecture Resource Usage of Fault Injected Circuits	48
3.4	Sensitivity of Fault Injected Circuits	49
4.1	Measured Resource Utilization and Clock Speed for Discrete Sizes of Integer Adders	59
4.2	Measured Resource Utilization and Clock Speed for Discrete Sizes of Integer Multipliers	61
4.3	Measured Resource Utilization and Clock Speed for a 4-to-1 Multiplexer	64
4.4	Measured Resource Utilization and Clock Speed for an 8-to-1 Multiplexer	64
4.5	Computed vs. Actual Values for Simple Add/Multiply Circuit	67
4.6	Resource Utilization for Multiplexed Add/Multiply Circuit	68
4.7	Resource Usage for the Circuit Shown in 4.17	68
4.8	Resource Usage Data for Four Versions of a Floating-Point Multiplier	70
4.9	Comparisons Using RALFs for the Multipliers from Table 4.8	71
4.10	RALF Comparisons for an Adder and a Multiplier	71
4.11	ASAP and ALAP Schedules for the SAM Problem Shown in Fig. 4.18	74
4.12	Node Distribution by Type for the DFG Shown in Fig. 4.19	75

List of Figures

Figure Page

2.1 Example FPGA-style LUT and routing network. 5

2.2 Continuous planning partitions. 14

2.3 Iterative Repair flowchart. 15

2.4 The Simulated Annealing algorithm. 16

2.5 The Genetic Algorithm. 17

2.6 The Stochastic Beam Search. 17

2.7 Hardware implementation of a Genetic Algorithm with short chromosomes [1]. 19

2.8 Simulated annealing pseudocode. An optimal solution is derived by repeatedly executing the five steps. 21

2.9 An example of iterative repair using simulated annealing. A solution is copied, altered, evaluated, compared against the current solution, and accepted conditionally. This process is repeated thousands of times to arrive at the optimal solution. 23

3.1 A dependency graph for 40 events. Events are represented by the numbered nodes. Edges indicate dependencies. Each event also uses one of four resource types, designated by the shape of the node. 25

3.2 Iterative repair architecture. A pipelined processor with associated memory constructs is derived from the simulated annealing pseudocode. 27

3.3 Multiport memories using BRAMs. In (a), four read ports use four BRAMs. In (b), four write ports use four BRAMs, but four clock cycles are needed to allow a write from each port. 28

3.4 Multiplexed connections between processing stages and memory modules for read accesses. Each memory block has four read ports. 29

3.5 Multiplexed connections between processing stages and memory modules for write accesses. Each memory block has one write port. 30

3.6 Method for passing memory block pointers between processing stages when (a) the solution in the Accept stage is NOT accepted and (b) the Accept stage solution is accepted. 30

3.7 Memory circuit that allows for double-wide data transfers in addition to word-sized accesses. 32

3.8 The Alter stage. 33

3.9 The dependency graph violation substage (DGV). 34

3.10 The total schedule length sub-stage (TSL). 35

3.11 The resource over-utilization sub stage (RO). 36

3.12 The Accept stage. 38

3.13 The Adjust Temperature module. 39

3.14 Best-case AMPS accelerator performance. 40

3.15 Worst-case AMPS accelerator performance. 41

3.16 System block diagram for partial reconfiguration of Evaluate module. . . . 44

3.17 Resource usage (LUTs) for 10 example problems. 45

3.18 Resource usage (flip-flops) for 10 example problems. 46

3.19 Resource usage (BRAMs) for 10 example problems. 46

3.20 Resource usage (DSP48s) for 10 example problems. 47

3.21 Power usage for 10 example problems. 47

3.22 Execution times for 10 example problems on both custom hardware and PowerPC 750. 49

3.23 Hardware vs. software results comparison. 50

4.1 Typical steps for hardware and software design. 52

4.2 Block-level diagram for translating simulated annealing C code to hardware. 54

4.3 Example of source C code and comparable 3-address single assignment code. 55

4.4 Obtaining a CDFG from 3-address single assignment code. 56

4.5 Optimization of a CDFG. 57

4.6 Optimization of a more-complex CDFG. Loop unrolling and predicative execution are represented. 58

4.7 LUT and flip-flop consumption for integer adders. 60

4.8 Maximum clock frequency for integer adders. 61

4.9 LUT utilization for integer multiplier. 62

4.10 Flip-flop utilization for integer multiplier. 63

4.11 DSP48 utilization for integer multiplier. 63

4.12 Maximum clock frequency for integer multiplier. 64

4.13 LUT utilization for 4-to-1 multiplexers. 65

4.14 LUT utilization for 8-to-1 multiplexers. 66

4.15 Simple circuit with an addition operation followed by a squaring operation. 66

4.16 Multiplexed add/multiply circuit. 68

4.17 An example architecture from a simulated annealing Alter stage. 69

4.18 Possible schedules and mappings for a simple DFG. The DFG shown in (a) can be scheduled and mapped as shown in (b) or (c). 73

4.19 A typical evaluate stage DFG for evaluation by the SAM algorithm. 75

4.20 Example of a DFG (a) with an associated architecture (b), including a multiplexer and three delay registers. 77

4.21 SAM execution flowchart. 82

4.22 Conversion of HIF to VHDL. 83

5.1 Example traveling salesperson problem for ten cities. The edges represent two possible orders of visitation. 84

5.2 Solution format and alter strategy for TSP. 85

5.3 An adjacency graph for GCP consisting of 20 regions. 86

5.4 Solution format and alter strategy for GCP. 87

5.5 Example of a dependency graph. Nodes represent tasks and edges represent dependencies. 88

5.6 Comparative performance in total execution time. 89

5.7 Speedup of SATH-generated circuits comparing total execution time with respect to other implementations. 90

5.8 Comparative performance in cycles per iteration. 90

5.9 Speedup of SATH-generated circuits comparing clock cycles per iteration with respect to other implementations. 91

5.10 Comparative LUT usage of circuits generated using Impulse and SATH. . . 92

5.11 Comparative flip-flop usage of circuits generated using Impulse and SATH. 92

5.12 Comparative DSP48 usage of circuits generated using Impulse and SATH. . 93

5.13 Comparative BRAM usage of circuits generated using Impulse and SATH. . 93

5.14 Comparative RALF usage of circuits generated using Impulse and SATH. . 94

Chapter 1

Introduction

In this introductory chapter, the motivation for deriving application specific hardware from a software source is discussed. Challenges that complicate the process are presented. Contributions of this research work are enumerated and an overview of the content of the remaining chapters is also included.

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) are becoming increasingly popular as a platform of choice for spacecraft computer systems. FPGA-based designs are much cheaper and have a shorter development cycle than traditional Application-Specific Integrated Circuits (ASICs), and provide more computing power and efficiency than standard microprocessors. Some of the current and planned space missions and experiments that utilize FPGA technology include MARTE (Mars Astrobiology Research and Technology Experiment) [2], the Discovery and New Frontier programs [3], the Dependable Multiprocessor [4], the Venus Express mission [5], and NASAs DAWN mission [6].

Simulated annealing is a widely used heuristic algorithm to solve challenging optimization problems. While it has been used extensively for static time design optimization, there is an increasing need to deploy such solvers in real time embedded systems. For example, NASA uses the CASPER [7] and ASPEN tools [8, 9] to design code for Iterative Repair, a simulated annealing algorithm that derives complicated event schedules on-board spacecraft. The complexity of these algorithms can be daunting for space based computers, which are significantly slower than state of the art microprocessors.

Combining FPGAs with simulated annealing algorithms would greatly improve system performance. Unfortunately, hardware architecture design targeting FPGAs is much harder

and more time consuming than software design and is daunting for software engineers without expertise in VLSI design.

To mitigate this design flow barrier, a methodology for the automatic derivation, from source code of simulated annealing scheduling algorithms, of FPGA-based application-specific processors is presented. This methodology is termed SATH (Simulated Annealing to Hardware).

1.2 Principal Contributions

The aims of this work are as follows. Mainly, a method for the conversion of software source code into appropriate hardware circuits designed to accelerate simulated annealing algorithms is described. The importance of selecting a proper architecture template is discussed. A method for design-time approximation of required hardware resources is presented. The performance of simulated annealing circuits generated using the proposed methodology are compared with the performance of circuits generated using a commercially-available software-to-hardware conversion tool, with significant improvements in both execution time and resource usage attained across all test cases. The execution time of the custom circuits are also compared execution time of standard desktop and space-based processors. While this research is specific to deriving hardware to accelerate simulated annealing codes, the techniques presented are applicable to a broad range of problem domains.

1.3 Overview

An overview of related works is presented with critical analysis in Chapter 2. Chapter 3 discusses a hardware template, by means of an example, that is used as a basis for converting software to hardware. Chapter 4 details the SATH algorithm to derive custom processors. Chapter 5 provides results comparing the performance of SATH-generated circuits and compares with that of traditional space-based processors and existing C-to-gates tools. Chapter 6 summarizes the research and provides directions for future work.

Chapter 2

Background

This chapter discusses recent advances in the different areas that are applicable to this research work. Topics of interest include FPGAs, C-to-hardware compilers, high-level synthesis techniques, and simulated annealing.

2.1 FPGAs

An FPGA is a silicon device that is in some ways comparable to an Application-Specific Integrated Circuit (ASIC). The vast majority of large-scale integrated devices used today in industry are ASICs, meaning that they are permanently configured for a specific application. An FPGA, on the other hand, consists of several static random access memory (SRAM) based reprogrammable logic blocks along with reprogrammable inter-block connections, allowing a single FPGA to be reprogrammed and used in a variety of applications. While FPGAs have the benefit of being dynamically reconfigurable, they are also larger, slower, and more power-hungry than ASICs. As FPGA technology continues to improve, they will command an increasing share of the integrated circuit market. Much research is currently being done on FPGA methods and applications. In addition to thousands of logic blocks, modern FPGAs from Xilinx [10,11] and Altera [12] have rich and powerful computing fabrics with complete with embedded ASICs such as multiply-accumulate units, block RAMs, and digital clock managers.

The basic building block of a traditional FPGA, as described in [13], is the lookup table (LUT). Modern FPGAs from Xilinx consist of thousands of four-to-one [10] or six-to-one [11] LUTs. A LUT is a memory element. Input lines represent an address. The output line provides the data stored at that address. Thus, any logical function that consists of four or fewer inputs and a single output can be mapped onto a LUT. Multiple LUTs are

Table 2.1: Truth Table for Eq. (2.1)

Inputs				Output
a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

employed to support more-complex functionality. As a simple example, consider the logical expression described by Eq. (2.1).

$$f = (ab + cd) \tag{2.1}$$

This function will fit in a single LUT, as there are four inputs and one output. The truth table for this function is shown in Table 2.1. The truth table is a direct representation of memory addresses and memory contents. All LUTs also have an associated flip-flop, which can be employed if desired, allowing both synchronous and asynchronous designs. LUTs communicate using programmable routing networks, which are configured to allow appropriate signal flow through the circuit. Fig. 2.1 shows a close-up of a typical four-to-one LUT with associated logic, and additionally an example of how multiple LUTs might be configured in a routing network. Because of the reprogrammable nature of both the LUTs and the routing network, an FPGA can be configured to create a vast number of circuits.

2.2 C-to-Hardware Compilers

Converting C code into a hardware specification has been a much-researched area over

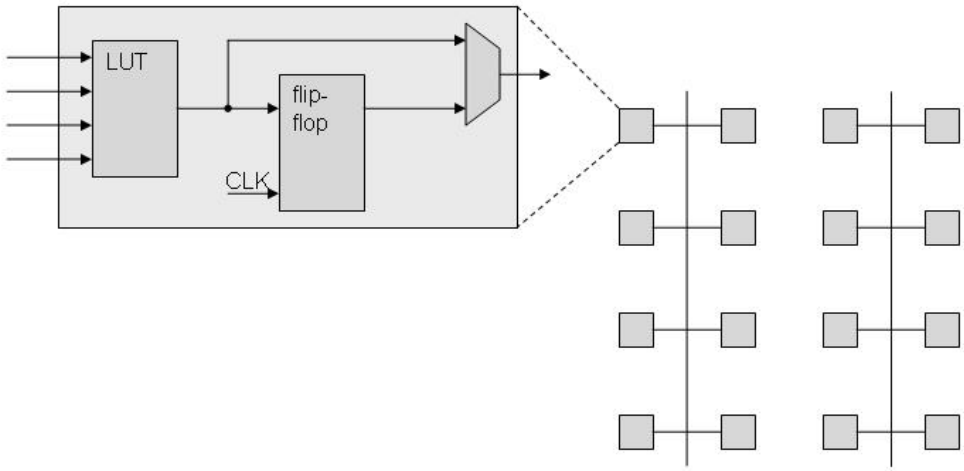


Fig. 2.1: Example FPGA-style LUT and routing network.

the past two decades. Several tools have been produced over this time period, each of which targets specific areas of the software to hardware conversion. In [14] a comprehensive summary of the different projects is provided. A summary of each of these tools is provided here.

The first attempt at a C to Gates tool was Cones [15], which was developed in 1988 at AT&T Bell Laboratories. Cones works on a subset of C with the introduction of some additional directives to facilitate translation to hardware. Cones translates C functions into combinatorial blocks, based upon the premise that a function consists of several inputs that influence one output. Arithmetic and logic statements are reduced through the use of Karnaugh Maps and similar techniques. Cones also unrolls simple loops. Cones cannot handle pointers, nested loops, recursive function calls, or dynamic memory allocation. Additional syntax is introduced to specify input and output data for each C function.

HardwareC [16] was developed at Stanford University in 1990. It is based upon the same syntax as conventional C, but includes custom hardware semantics. HardwareC additions include both procedural and declarative semantics, which means a design can consist of either sequences of operations or as a structure of interconnecting components. HardwareC models hardware as concurrent processes with inter-process communication facilities. The level of parallelism can be specified by the programmer to be sequential, data-parallel, or

parallel for a given design. Lastly, HardwareC supports constraint specifications, where time and resource constraints can be imposed.

Transmogrifier C [17] from the University of Toronto was released in 1995. It is another variation on a subset of the C programming language. Integer addition and subtraction are supported. Compiler directives are used to specify data lengths. If statements, while loops, and function calls are also supported. The compiler does not support multiplication, division, pointers, arrays, structures, recursion, or floating-point arithmetic.

In 2002 SystemC [18] was introduced. SystemC is actually a C++ library. Classes are defined for simple Verilog constructs such as combinatorial and sequential modules. Simple, user-specified concurrency is allowed through the use of threads. SystemC is primarily a simulation language, although a restricted subset can be synthesized. Similarly, Ocapi [19] and PDL++ [20] are also built on C++. Proprietary classes are provided for creating finite state machines and data paths. The C++ code is translated to VHDL or Verilog for synthesis and design implementation.

One of the most successful ventures at translating C code to hardware was C2Verilog [21], which was introduced in 1998. C2Verilog supports standard ANSI C code. The programmer is not required to provide any hints or directives on concurrency or partitioning. The entire C language is supported, including pointers, recursion, floating-point data types, arrays, structures, etc. C2Verilog works by performing global analysis on a program and creating always blocks and concurrent statements. Functions are represented using state machines. Pointers and dynamic memory allocation are managed by creating a dedicated RAM of sufficient size and sufficient IO ports to allow for efficient access.

Cyber [22], released by NEC in 1999, is yet another subset of the C programming language. The C subset is once again augmented with bit-length and input/output port declarations, as well as data-transfer types such as registers, terminals, latches, and tri-state buffers. The programmer must also specify code sections that are synchronous, asynchronous, or concurrent. Typically, difficult constructs such as recursion, dynamic memory allocation, and pointers are not supported. This augmented subset of C is termed BDL

(Behavior Description Language). Cyber is actually a behavioral synthesis system that takes behavioral code in BDL or VHDL and produces synthesizable register transfer logic.

Handel-C [23], developed by Celoxica, is a widely-used C variant. Handel-C is built upon the CSP [24] algebra for modeling process concurrency. Data widths can be specified by the programmer. The programmer can designate code sections as being concurrent. Input and output ports are introduced to allow an interface with the outside world. Synchronous message massing between processes is available. A variation on the basic C switch statement generates hardware multiplexers. Every instruction in Handel-C takes a single clock cycle. Bach C [25] is almost identical to Handel-C, providing the same explicit concurrency and message-passing capabilities. Pointers are not handled in either Handel-C or Bach C.

Another C variant is SpecC [26]. SpecC provides a set of 33 key words as additions to the ANSI C language that specify how the compiler should create finite state machines, concurrency, pipelining, etc. Once again, complicated features such as recursion and pointers are not supported.

The Trident C compiler [27] is a tool targeted specifically at floating-point operations. Concurrency and throughput are maximized through the use of pipelined floating-point units on an FPGA. Yet again, features such as dynamic memory allocation and pointer manipulations are not supported.

Another work worthy of mention is SPARK [28]. SPARK translates a subset of ANSI C to register-transfer level VHDL. Much work is done on the analysis of loops and conditional branches. Speculative code execution is also employed, meaning that computations performed within a conditional block are commenced before it has been determined that the block will be entered. Compile-time analysis is performed to determine what the best instructions are for speculative execution. The SPARK compiler cannot handle pointers or dynamic memory allocations.

Another system of note is CASH [29]. CASH, or Compiler for Application-Specific Hardware, differs from all previously mentioned systems because it generates asynchronous

hardware. Starting from a pure ANSI C code, CASH identifies instruction level parallelism and generates asynchronous data-flow circuits that support these parallel constructs.

One of the most-recent players in the C-to-gates market is Impulse [30, 31]. Impulse claims to be a true subset of C. It consists of a C library which provides a functional interface for mixed-signal design. It does not support nested function calls. Some instruction-level parallelism can be explicitly identified by the programmer.

To summarize this discussion on work in the area of C-to-architecture translation, compiling pure C code into gates is a difficult task. Common areas of difficulty include recursive function calls, dynamic memory allocation, pointer manipulation, support for all C data types (including floating point representations), and detection of concurrency. In [14] a detailed discussion of the shortcomings of using C to derive hardware is presented. Specifically, the concepts of concurrency, timing, data types, and communication can not be specified using standard C. These issues are generally resolved by either reducing the input language to a restricted subset of C, or by introducing compiler directives or other keywords or labels to indicate how the compiler should proceed. The ideal situation is to start with fully compliant ANSI C source code, thus allowing for the translation of existing code directly into hardware without modification. Of all the projects discussed, only C2Verilog claims to support the entire ANSI C standard with no restrictions or additions. This means that the C2Verilog compiler is responsible for all extraction of parallelism. While the C2Verilog compiler can identify instruction and loop-level parallelism, it cannot extract process-level parallelism. The ability of the compiler to recognize parallelism is directly dependent upon the syntax of the source code source codes written with concurrency in mind generally result in better hardware architectures than those that are written for conventional sequential machines. In addition, current C-to-hardware tools fail to take into account several important hardware design factors, including available FPGA area, power consumption, execution speed, and fault tolerance, all of which are critical to engineering in the space environment. In Chapter 5, results obtained from the Impulse tool are compared with the novel architecture generation approach described in this research.

2.3 High Level Synthesis Techniques

High level synthesis (HLS) in the context of FPGA-based architectures is a powerful tool. HLS is, in essence, utilizing one or more search algorithms to derive an efficient hardware architecture that can support an algorithm specified in high-level code. The goal is to identify the architecture that yields the best tradeoff between conflicting goals, such as FPGA area usage versus system throughput. The number of possible architectures is generally very large, thus demanding an intelligent search method to arrive at a solution within a reasonable amount of time. Common techniques for performing HLS include integer linear programming, Markov decision processes, Pareto optimality, and dynamic programming, well as heuristic searches such as simulated annealing, genetic algorithms, tabu search, and design-space pruning. Heuristic scheduling algorithms are also applicable to this discussion. A sampling of some of the different techniques used for HLS is discussed in this section.

As part of HLS, a target processor type must first be selected to guide the synthesis. An overview of the different types of processors that are typically considered is provided in [32]. Reduced Instruction Set (RISC), Complex Instruction Set (CISC), VLIW (Very Long Instruction Word), dataflow, tagged-token, and pipelined architectures are all commonly utilized. An HLS algorithm is generally restricted to one flavor of processor in order to put an upper bound on the time needed to search the architectural space. Trying to search across all possible architectures is considered to be an intractable problem.

Generally used heuristics for HLS include the comparable techniques of simulated annealing, genetic algorithms, and tabu search. A study has been performed which compares the three methods, arriving at the conclusion that tabu search may be better in some instances [33]. In [34], a good description of performing HLS for a reconfigurable processor is described. Important elements to be considered in the architecture space include allocation of computational, control, and memory resources, along with the scheduling of operations onto these resources. Exploration can occur in both parallelization (spatial optimization) and pipelining (temporal optimization). Simulated annealing is employed as the heuristic

search method. Over thousands of iterations of the simulated annealing algorithm, the throughput of the processor gradually improves.

An HLS using genetic algorithms is given in [35], which details the use of a genetic algorithm for deriving a custom architecture for a digital camera. The processing platform consists of a PowerPC core, data cache, instruction cache, memory, and buses. Different architectures are derived depending upon the file format and photo resolution.

In [36] an improvement to the basic genetic algorithm is proposed. A genetic algorithm maintains a population of current solutions at any given time during execution. A technique called fuzzy clustering is introduced which combines solutions into clusters or groups based upon score. The GA can then discard lower-scoring clusters and focus on the more-promising ones. This technique is once again applied to deriving VLIW architectures for video processing.

A project is presented in [37] in which an iterative improvement algorithm (based upon simulated annealing) is utilized to design processors for noise cancellation algorithms. The resulting processor is a VLIW processor with an arbitrary mix of multipliers, adders, and multiply-and-accumulate units.

Additional tools that use simulated annealing coupled with the concept of Pareto-optimality are presented in [38] and [39]. A solution is said to be Pareto-optimal if there does not exist a solution that betters one parameter without worsening one or more of the others. These Pareto-optimal solutions can be used to guide the search of the simulated annealing algorithm. While a Pareto-optimal solution may not be a globally optimal solution, it is a good candidate for an area in which to focus the search. This technique was applied to a MIPS processor platform with adjustable data cache, instruction cache, and main memory sizes. Typical benchmarks in signal processing and image conversion were used to test the system.

In [40], a system has been developed to explore the space of heterogeneous micro-architecture processors, 20 to 50 of which may reside on a single FPGA. The search tool uses Integer Linear Programming as the search method, where the goal is to maximize

throughput. Integer Linear Programming is a method for solving a system of linear inequalities. The linear constraints define a polyhedron, whose edges can be traversed until the optimal solution is found. Integer Linear Programming is an NP-complete algorithm that is often coupled with branch-and-bound techniques to reduce the compute time. The tool was tested by implementing an FPGA-based IPv4 packet forwarder that can outperform a hand-tuned design.

Another Integer Linear Programming HLS system is presented in [41]. ILP is used to determine the most profitable extensions that should be added to a base processor for various data encryption and decryption techniques for high-bandwidth data. Extensions can include additional arithmetic units or combinations of units, such as a multiply-accumulate unit. The processor is targeted for implementation on an ASIC and the typical constrained optimization problem of throughput versus area utilization is solved.

The inventors of SUIF have also made significant inroads in the area of FPGA-based HLS, specifically targeting source code that consists of multi-dimensional array accesses [42]. Concepts from SUIF have been combined with ideas from C-to-architecture tools to develop a method for exploring the time/space tradeoff in a custom FPGA architecture. The method involves the use of hardware synthesis tools from Xilinx or Altera as a mechanism for providing timing and area estimates for a potential design. The design is revised over many iterations of trial-and-error until an acceptable (but not necessarily optimal) architecture is discovered.

One technique for decreasing the time needed to find an architecture is a technique known as design space pruning [43]. Essentially, this method provides early estimates of area/latency tradeoffs to the search engine, immediately eliminating any solutions that are estimated to be poor performers by bounding the value of acceptable solutions. The searcher can then focus on optimizing more promising solutions. This technique can be successfully employed in combination with a standard exhaustive search, or with one of the other heuristic searches. Estimations are performed by modeling architecture efficiency based upon number of data-path operators, data bit-widths, register file size, number of

control units, control signal complexity, total memory size, and number of read and write ports needed to support concurrent memory accesses. The resultant architecture would generally be a VLIW-style processor.

Another technique for HLS is to model the search problem as a Markov Decision Process (MDP) [44]. An MDP is a flavor of reinforcement learning in which a program traverses design states in a decision tree probabilistically according to values that have been learned over time. In other words, an MDP is initially a poor-performing random search. However, over thousands of trials, the MDP can be trained to produce high-quality architectures. This algorithm has been applied successfully to derive custom VLIW processors for various image and video compression algorithms.

When the HLS problem consists of a traditional load/store processor that needs to be streamlined for a specific application, a technique such as CUSTARD [45] can be used to design an efficient multi-threaded processor. CUSTARD begins with a traditional MIPS integer pipeline processor. The processor is customized in three ways. First, unneeded instructions are removed from the instruction set. Second, additional pipelines are introduced to maximize parallelism. Each pipeline need not be identical; rather, each can be customized to support only the needed instructions. Third, the simple instructions supported by the MIPS RISC architecture can be combined to form more complex instructions (combining multiplication with addition to form a multiply-accumulate instruction is one common example). The cost of introducing complex instructions must be evaluated carefully, as they sacrifice versatility for speed. A cycle-accurate simulator is used to measure the performance of the candidate processors.

Heuristic scheduling methods such as List Scheduling [46] and Force-Directed Scheduling (FDS) [47, 48] also play a part in the HLS discussion. List Scheduling attempts to minimize execution time by finding the best schedule of a dataflow graph given a set of resources. Force-directed scheduling attempts to derive the smallest set of resources needed to schedule a dataflow graph within a fixed execution window. Neither method takes care of actually mapping graph nodes to resources; thus timing and routing overheads (regis-

ters and multiplexers) are ignored. While these techniques are useful for scheduling small graphs, they are not optimal schedulers. Variations to the basic List Scheduling algorithm, such as Modified Critical Path [49,50], Earliest Time First [51], Dynamic Critical Path [52], topological clustering [53], Critical Node Parent Trees [54], Cone-Based Clustering [55], and Partial Critical Path scheduling [56], have been proposed over the years. These algorithms improve the performance of the basic List Scheduling algorithm at the expense of increasing algorithm complexity. Modifications to the FDS algorithm include one which combines scheduling with mapping into a schedule-and-map (SAM) algorithm [57].

In summary, HLS is a powerful tool. Many search strategies exist and many applications can be targeted. The tradeoffs between a heuristic approach such as simulated annealing (faster results) and an exhaustive approach such as integer linear programming (correct results) must be weighed carefully. HLS techniques can take place at different design levels. Discrete components such as processors and memories can be combined in different ways, or the internals of the processor itself can be customized.

2.4 Iterative Repair and Simulated Annealing

ASPEN (Automated Scheduling and Planning Environment) [9] and CASPER (Continuous Activity Scheduling, Planning, Execution, and Replanning) [7] are tools that were developed at the Jet Propulsion Lab for use in modeling and implementing space-based mission planning and scheduling algorithms. ASPEN consists of a GUI-based design environment that supports a C-like programming language for modeling events that must be scheduled. CASPER is a stripped-down version of ASPEN that was designed to fly on the satellite, performing dynamic planning and continuous rescheduling of mission-critical events in real time. CASPER continuously runs an Iterative Repair algorithm to constantly improve and update the schedule. In traditional planning methods, events are labeled as either in view or beyond the horizon. Only those events that fall within the event horizon are planned. Continuous planning takes a different approach. Rather than utilizing discrete event horizons, all tasks to be scheduled are available to the planner at all times. The tasks are divided into short-, medium-, and long-range types, where short-range tasks are the

most detailed in terms of specific start-times, exact event durations, resource utilizations, etc., while long-range task representations are very general, rough estimates of timing and resources needs. This organization is shown in Fig. 2.2. As time progresses, tasks pass

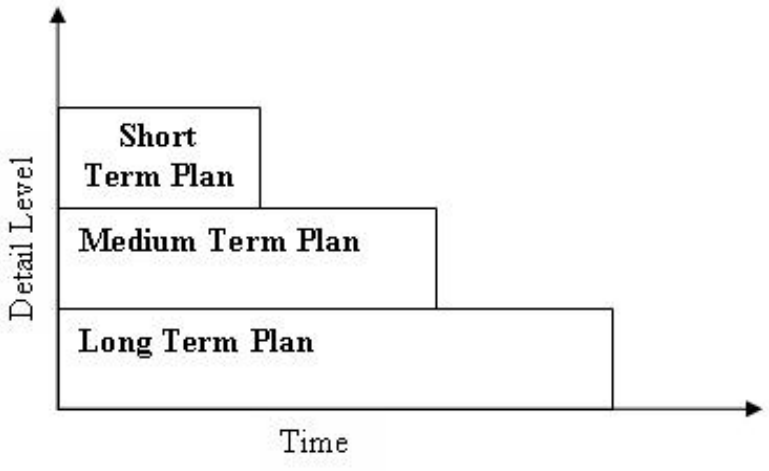


Fig. 2.2: Continuous planning partitions.

from one planning frame to another, depending upon time left until execution. The planner runs constantly to improve the schedule. The Iterative Repair algorithm runs as detailed in Fig. 2.3. A first-guess solution is generated initially which satisfies timing constraints while ignoring resource constraints. This solution is then gradually improved, or repaired, over many iterations until an optimal (or close-to-optimal) schedule has been found. These improvements are made by either reassigning an event to an unused resource or moving the event to a different time slot.

Iterative Repair is, by nature, a greedy algorithm. Schedules can be improved to a point, but the algorithm can become trapped in a local optimum in the search space. One or more inferior solutions may need to be used as stepping stones to arrive at the global optimum. This technique of sacrificing local optimally for the global good is employed by Simulated Annealing. Simulated Annealing is a method for performing combinatorial optimization on a large search space where an exhaustive search is not tractable. Simulated Annealing is based upon the metallurgic phenomenon of annealing, in which a metal is heated to an extremely high temperature and then allowed to cool slowly, resulting in

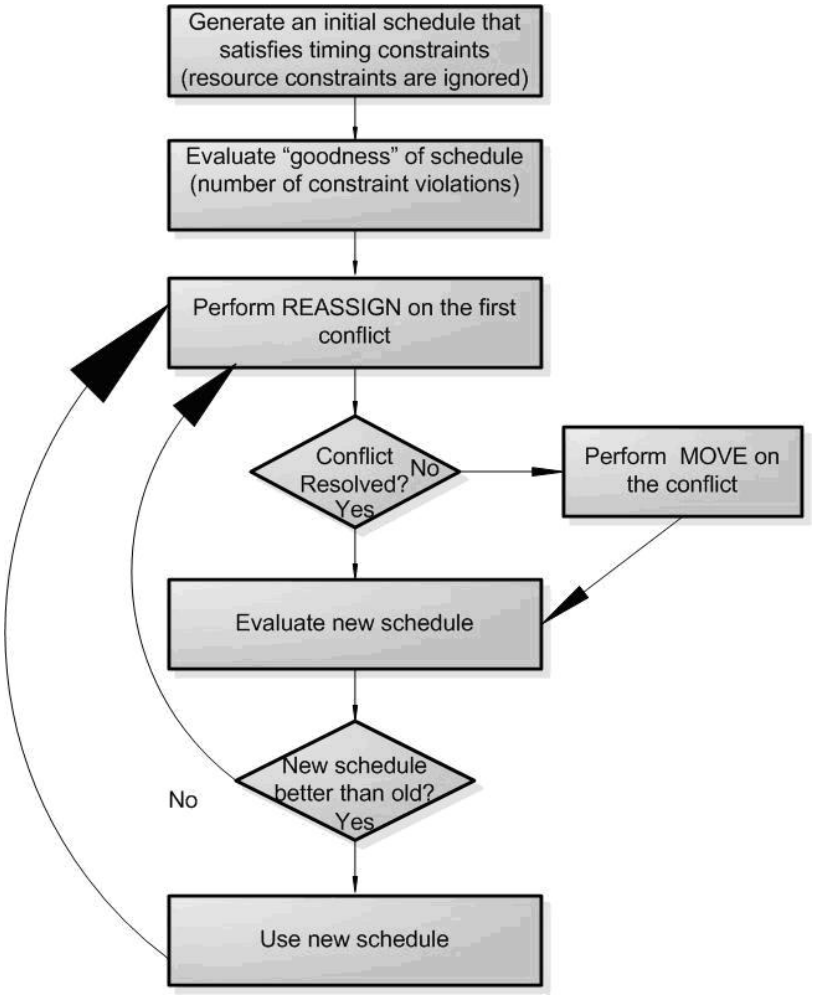


Fig. 2.3: Iterative Repair flowchart.

a (near) minimal energy configuration of the crystalline structure. Simulated Annealing takes an initial solution and minimizes the energy of the solution through successive steps, in each of which a slightly different solution is compared to the current solution. In order to avoid becoming trapped in local optima, solutions with higher energies are accepted probabilistically, based upon the current temperature of the system. The temperature is gradually lowered over many iterations until an optimal solution is found.

As discussed above, the Iterative Repair algorithm for event scheduling utilizes the Simulated Annealing heuristic to effectively search for optimal solutions. Other common heuristic searches which yield similar results and which could be used in place of Simulated Annealing include Genetic Algorithms [58] and Stochastic Beam Search. A summary of all three methods is provided in [59]. Pseudocode for the three algorithms is presented below in Fig. 2.4, Fig. 2.5, and Fig. 2.6. All three algorithms operate on variations of the

Simulated Annealing

```
generate initial solution
evaluate initial solution
set initial temperature
Loop
      generate new solution
      evaluate new solution
      set initial temperature (T)
      If new better than old
            overwrite old solution with new
      Else
            compute ΔE (ΔE = |old score - new score|)
            compute acceptance probability (p = exp(ΔE/T))
            generate random number between 0 and 1
            If random <= acceptance probability
                  overwrite old solution with new
            EndIf
      EndIf
      lower T
EndLoop when T ≈ 0
```

Fig. 2.4: The Simulated Annealing algorithm.

same general premise: gradually improve a solution (or set of solutions) over time until a solution of sufficient quality is discovered. Simulated Annealing (SA) operates on a single solution, while Stochastic Beam Search (SBS) and Genetic Algorithm (GA) maintain a

Genetic Algorithm

```
generate k initial solutions
evaluate k initial solutions
Loop
      Loop k times
            select random solutions x and y from k
            crossover x and y to form new solution
            generate random number between 0 and 1
            If random <= mutation probability
                  mutate new solution
            EndIf
            evaluate new solution
            add new solution to pool
      EndLoop
      select k best solutions out of 2k available
EndLoop when improvement ceases
```

Fig. 2.5: The Genetic Algorithm.

Stochastic Beam Search

```
generate k initial solutions
evaluate k initial solutions
Loop
      generate k solution
      evaluate k solution
      merge old and new solutions for 2k solutions
      select k best solutions
EndLoop when improvement ceases
```

Fig. 2.6: The Stochastic Beam Search.

pool, or population, of solutions (solutions are also termed chromosomes in GA). All three algorithms utilize similar subroutines for evaluating solutions and generating new solutions, although the crossover and mutation operators used in GA are a bit more complicated. All three algorithms also depend heavily on random number generation and probabilities. GA and SBS differ from SA in the technique used for avoiding entrapment in local optima. In SA, suboptimal moves are accepted as long as the temperature is sufficiently high and the value of the new solution is reasonably good. GA and SBS solve the problem by maintaining a pool of current solutions such that it is statistically impossible for all solutions to fall in an area around the same local optima.

In theory, implementing combinatorial search algorithms in hardware could significantly speed-up the search process. Large amounts of parallelism and pipelining can be extracted from SA, GA and SBS, since deriving a new generation is largely only a function of the previous generation. Hardware-based GA implementations abound in the literature. Some recent examples of FPGA-based GAs are discussed here.

A GA has been implemented on an FPGA for the purpose of blind signal separation [1]. Function-level pipelining has been implemented. The high-level flow chart, which is almost identical for all hardware GAs surveyed, is shown in Fig. 2.7. This system was implemented on a Xilinx Virtex FPGA, using a chromosome length of 64 bits and a population size of 80 chromosomes. As long as the chromosome length is kept reasonably small, this type of technique in which entire chromosomes are passed between pipelined modules works well. GAs are also commonly used for filter design. In [60], a GA for grey-scale filter design has been implemented on a Xilinx Spartan II FPGA. A pipelined approach similar to the one shown in Fig. 2.7 is once again taken. Chromosomes are 16 bits in length and a generation consists of 10 chromosomes. Another example is using a GA to perform function interpolation [61]. In this case, chromosomes are 24 bits in length or less. A system for maximizing algebraic expressions has been developed with 5-bit chromosomes [62]. Three different sets of libraries and design algorithms written in both Verilog and VHDL for the implementation of various types of GAs with relatively small chromosome lengths have also

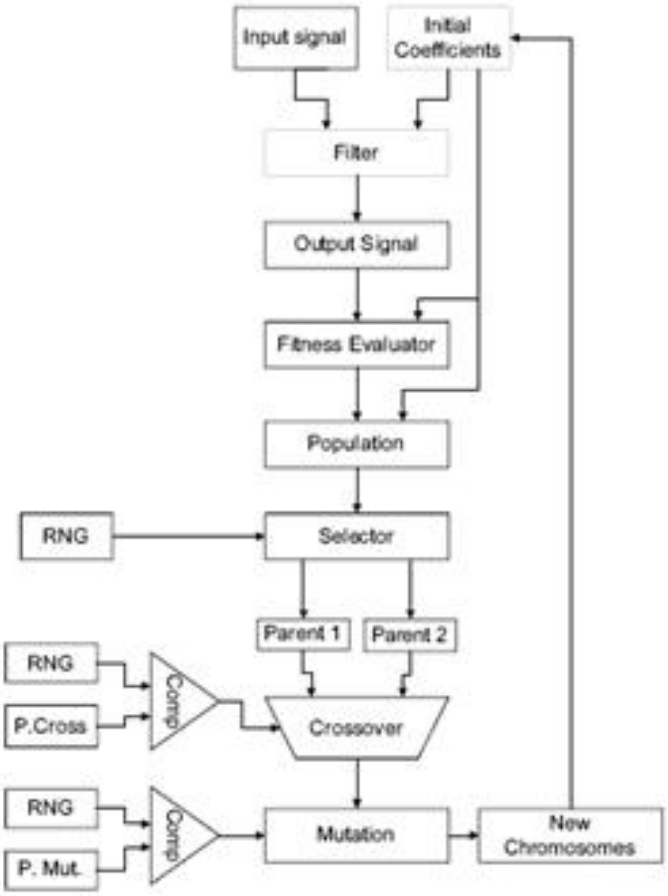


Fig. 2.7: Hardware implementation of a Genetic Algorithm with short chromosomes [1].

been developed [63–65].

The examples discussed above are very simple, as the chromosome length in all cases is so small that the entire chromosome can be passed between pipelined modules simultaneously. Most real-world problems, however, are complex enough that a chromosome can be hundreds of bytes in length. The simple pipelined implementations discussed above clearly cannot handle this complexity, as 100-byte wide busses are not feasible for implementation on an FPGA.

One method for resolving this problem is to split the chromosome into manageable chunks and transfer one chunk of data on each clock cycle [66]. This allows a modified version of the simple pipelined architecture to be created. Obviously, as the ratio between chromosome length and bus width grows, the time used in data transfers also increases.

Traveling Salesperson is the classic combinatorial search problem, the goal being to find shortest path for visiting every node in a set exactly once. A version of the problem has been solved using GA on a Xilinx Virtex-E FPGA [67]. Rather than coding modularly in VHDL or Verilog, the implementation was performed using Handel-C. Population size is 200 chromosomes, with the chromosome size being variable. Explicit pipelining is not realized in this design, as Handel-C is a behavioral language rather than a structural language. Any parallelization is specified by the programmer and interpreted by the Handel-C compiler.

An example of performing SBS in hardware was used for speech recognition algorithms [68]. Chromosome widths in this example are once again limited to a few bits. Because of the inherently serial nature of the SA algorithm, it is not generally considered an interesting problem for parallelization or hardware implementation.

The chromosomes needed for scheduling are much larger than what can be transmitted on a bus in one shot. For example, take a system in which 100 tasks need to be scheduled within a 24 hour period. If the tasks need to be scheduled with a resolution of 1 minute, the chromosome would consist of 100 11-bit numbers. An 1100-bit chromosome is much too large to transfer as a single chunk. Novel architectures must be developed to accelerate this algorithm.

A hardware implementation of the FPGA place-and-route algorithm using Simulated Annealing has been done [69]. The place-and route algorithm searches for a layout of a circuit on an FPGA, including the use of both logic blocks and routing resources, that minimizes resource utilization while maximizing circuit performance. The architecture in this case is based upon systolic arrays, rather than the pipelined structure advocated in this paper. Comparable speedups approaching three orders of magnitude were found using this architecture. As Simulated Annealing is the heuristic search employed by Iterative Repair,

```
temperature ← INITIAL_TEMP
generate initial solution
compute score of initial solution
while temperature > STOP_THRESHOLD
     copy: current_solution ← best_solution
     alter: modify current_solution
     evaluate: compute score of current_solution
     accept: current_solution better than best_solution?
     adjustTemperature
```

Fig. 2.8: Simulated annealing pseudocode. An optimal solution is derived by repeatedly executing the five steps.

a detailed description of the algorithm is now provided. As described in Fig. 2.4 and revised in Fig. 2.8, an initial solution is generated, usually randomly, and evaluated. This initial solution is designated as the current solution until a new one is accepted. The main loop is now entered, which generally loops several thousand times. On each iteration, the current solution is copied verbatim to a second buffer, where it is designated as the next solution. This next solution is then altered slightly and evaluated. The score of this new solution is then compared against the score of the current solution. The crux of the algorithm is determining whether to accept the next solution as the new current solution or discard it in favor of the keeping the resident current solution. This decision is made according to Eq. (2.2).

$$p = e^{\frac{\Delta E}{T}}, \Delta E = S_{next} - S_{current} \tag{2.2}$$

In this equation, S_{next} and $S_{current}$ are the scores of the current and next solutions, respectively, and T represents temperature. The probability p is a function of both the

temperature and the difference between the score of the current solution and the score of the new solution (ΔE). A random number is generated and compared to p to determine whether a solution should be accepted. When the temperature is high, suboptimal solutions are more-likely to be accepted. This feature allows the algorithm to escape from local minima as it searches the solution space and zero in on the true optimal solution. The last step in the loop decreases the temperature according to a pre-determined schedule. A typical method is to geometrically decrease the temperature by multiplication by a cooling rate, which is generally a number such as 0.99 or 0.999. The closer the cooling rate is to 1.0, the more times the loop will execute. This results in longer program execution, but also improves the probability of finding the best solution. Cooling too fast reintroduces the local-optima entrapment problem to the system.

Fig. 2.9 shows how simulated annealing can be applied to an iterative repair problem. In this case, a simple example consisting of ten events is presented. These events are numbered 0 through 9, and can be treated as indices to a solution array in computer memory. At any point in time, two solutions are maintained: the current solution and the potential next solution. From Fig. 2.4, the first step in the loop is to copy the current solution into the next solution buffer. Once this is done, the next solution must be altered in some way. Fig. 2.9 depicts a simple value swap, where two events are selected at random and the respective start times are swapped. This new solution must then be evaluated to determine how it compares with the current solution. Factors to take into consideration when computing the value of a schedule could include effective resource utilization, dependency graph violations (when a child event is scheduled before a parent event), and overall length of the schedule. Once the new schedules value has been determined, Eq. (2.2) is applied to determine whether or not it should replace the current solution. If the next score is better than the current score, it replaces the current score unconditionally. If it is worse, it is accepted with the computed probability, depending on both the score of the solution and the temperature. The temperature is then updated as described previously. This process is repeated until the temperature falls below a predetermined threshold, at which time the best schedule found

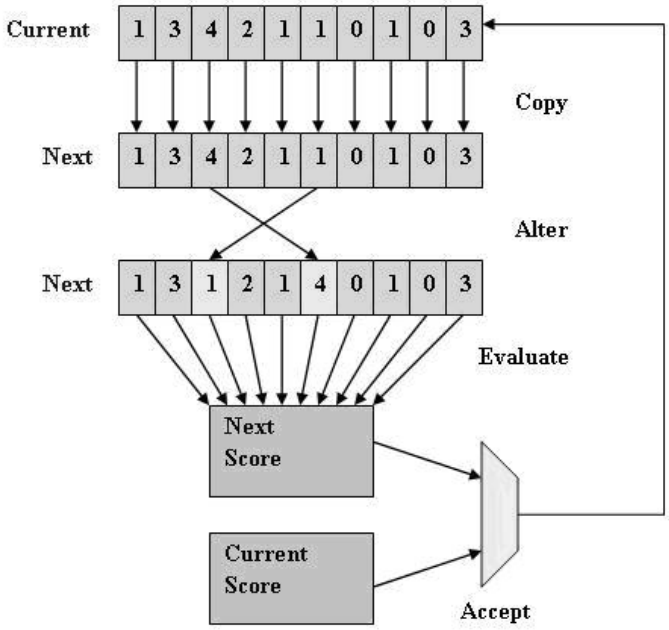


Fig. 2.9: An example of iterative repair using simulated annealing. A solution is copied, altered, evaluated, compared against the current solution, and accepted conditionally. This process is repeated thousands of times to arrive at the optimal solution.

is returned by the system.

2.5 FPGAs in the Space Environment

The space environment is not electronics-friendly. The sun is constantly spewing large amounts of fast-traveling, highly-charged particles into space in a phenomenon known as the solar wind [70]. These high-energy particles can affect electronic circuits in a variety of ways, causing single-event upsets (SEUs) and single-event latchups (SELs). SEUs occur when a transistor is energized by a high-energy particle, resulting in a bit flip. This phenomenon can occur in any part of a circuit, resulting in temporary data corruption, code corruption, or, in the case of an FPGA, even hardware architecture corruption. A SEL is similar to a SEU, occurring when a high-energy particle permanently damages a transistor rendering it unusable. Techniques have been developed for implementing fault-tolerant circuits on FPGAs. In [71] a summary is provided of current techniques for implementing fault tolerance on SRAM-based FPGAs.

For example, in [72], a system is specified for utilizing Triple Modular Redundancy (TMR) to provide fault tolerance against SEUs. In this scheme, the circuit is duplicated on three different FPGAs. A voter mechanism is employed to produce the result, assuming at least two out of the three FPGAs are operating correctly. An external microcontroller and radiation-hardened PROM complete the circuit. When voting is not unanimous, the microcontroller reprograms the faulty FPGA using bit-stream data read from the PROM. A similar paradigm, utilizing radiation-hardened FPGAs as both controllers and processors, is presented in [73]. Another work utilizing a radiation-hardened ASIC as the voter is described in [74].

Techniques have also been developed for recovering from SELs, which permanently damage the FPGA fabric. In this case, portions of a circuit residing in a damaged sector must be moved to a physically close intact and unused region. In [75] evolutionary techniques are used to determine how hardware blocks should be placed on an FPGA to facilitate optimal rearrangement. At design time, a genetic algorithm is used to determine the most flexible manner in which a circuit can be mapped onto an FPGA, creating simulated faults to observe needed patterns of rearranging.

Lastly, Xilinx provides a tool for the automatic introduction of TMR protection into an FPGA circuit [76]. The new circuit takes just over three times the area of the original, as the circuit is instantiated three times and voter circuitry is also employed. A Masters Thesis by Jeff Carver at Utah State University is in progress that discusses TMR and other novel methods for providing FPGA fault protection.

Chapter 3

Identifying a Hardware Template

A preliminary step in the development of a software-to-hardware conversion methodology is to determine the general form of the hardware architecture and test this form through manual implementation of an appropriate design. This step is critical to the derivation of the methodology, as the high-level hardware template can have a large influence on the architecture performance. In this section an analysis of the iterative repair algorithm from the perspective of a circuit designer is provided and a suitable hardware architecture framework derived from a software implementation of event scheduling using iterative repair is discussed. The source code used in this chapter can be found in Appendix A. Iterative repair specifics were discussed in Chapter 2. This framework will henceforth be classified as an AMPS accelerator circuit.

In the AMPS accelerator circuit framework, a solution is represented as a string of start times for v events numbered 0 to $v - 1$. Events have dependencies, meaning that certain events must complete before others can start. Each event also has an associated resource type. Fig. 3.1 depicts a possible event dependency graph. The shape of the event node designates the resource type. Each event takes one time step to complete. Additional input

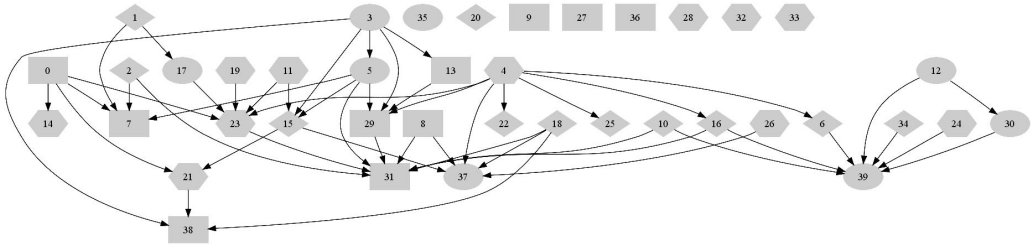


Fig. 3.1: A dependency graph for 40 events. Events are represented by the numbered nodes. Edges indicate dependencies. Each event also uses one of four resource types, designated by the shape of the node.

parameters needed are a maximum schedule length, an initial temperature, a cooling rate,

Table 3.1: Variable Definitions

Notation	Meaning
v	Number of events
E	Number of dependency edges
t_r	Number of resource types
n_{max}	Maximum allowable number of any resource type
L_{max}	Maximum allowable schedule latency
b	Number of storage bits available on one BRAM (18,432 bits)

and a termination threshold, all of which are provided as inputs to the design. Table 3.1 contains a list of variables to describe these parameters that are used throughout this chapter. Throughout the architecture, 16-bit integer arithmetic and 32-bit floating-point arithmetic are assumed. As shown in Fig. 3.2, the architecture is composed of a four-stage pipeline coupled with five memory banks. Each stage in the pipeline corresponds to a step in the simulated annealing pseudocode copy, alter, evaluate, and accept. A global controller coordinates execution and data exchange between the units. An interface between memory banks and processing stages is provided. An Adjust Temperature module controls the cooling process. As this is a pipelined architecture, it can only operate as fast as the slowest stage. To customize the framework for a specific problem size standard design practices can be employed in the more complex stages to minimize the latency and balance the pipeline.

3.1 The Memory Sub-System, Data-Routing Module, and Main Controller

The architecture consists of five memory modules, numbered zero through four in Fig. 3.2, derived from Xilinx FPGA BRAM blocks. Each memory module consists of one write port and four read ports, all 16 bits wide. Four read ports are needed to facilitate parallelism in the Evaluate stage. Although Xilinx BRAMs are comprised of two read ports and two write ports, we have found that VHDL code mapped on to a Xilinx BRAM using ISE permits only one read and one write port. Thus, a minimum of four BRAMs are used in the instantiation of each memory bank. If data size exceeds the 18k bits available in a BRAM, the BRAM usage doubles. Equations to describe BRAM usage are presented in

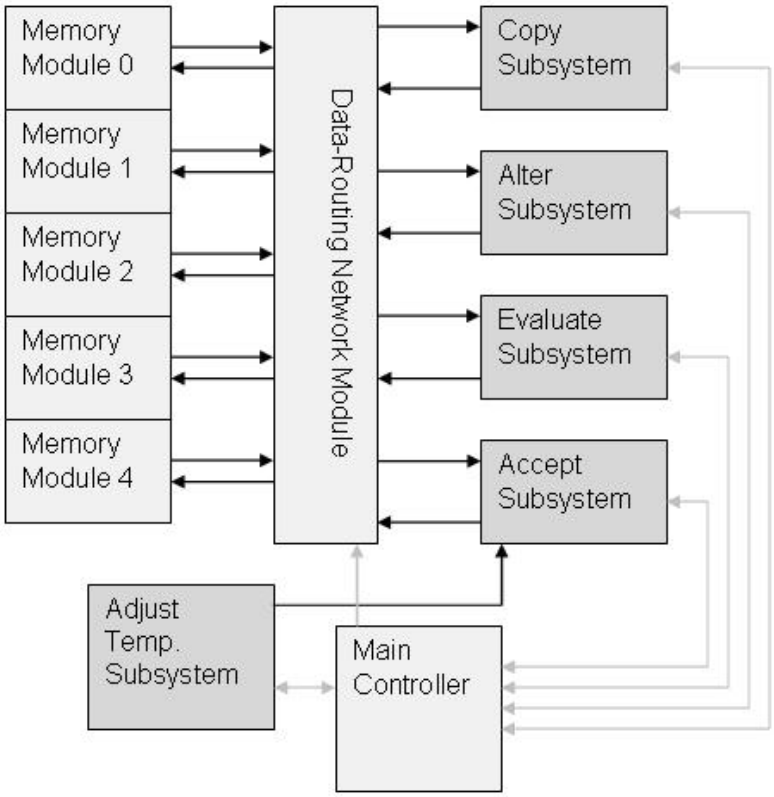


Fig. 3.2: Iterative repair architecture. A pipelined processor with associated memory constructs is derived from the simulated annealing pseudocode.

the Architecture Summary section. Fig. 3.3 shows how multiple BRAMs can be combined to build a multi-port memory. In Fig. 3.3a, four BRAMs are used to create a memory with four read ports. Each BRAM holds an identical copy of the data. In Fig. 3.3b, an attempt is made to build a memory with four write ports. Once again, each BRAM holds identical data. Unfortunately, four writes cannot occur simultaneously. The only work-around for this issue is to clock the memory at four times the rate of the system clock, resulting in the illusion of simultaneous writing. This effectively decreases the maximum clock frequency of the circuit by a factor of four, thus negating any gains that could be achieved through four parallel writes. Because of this, the memory blocks shown in fig. 11 are limited to one write port each. Each memory bank holds a solution and the score of the solution. At any point during execution, one memory bank is associated with each of the four processing stages in the pipeline. The remaining memory block holds the current solution. The main

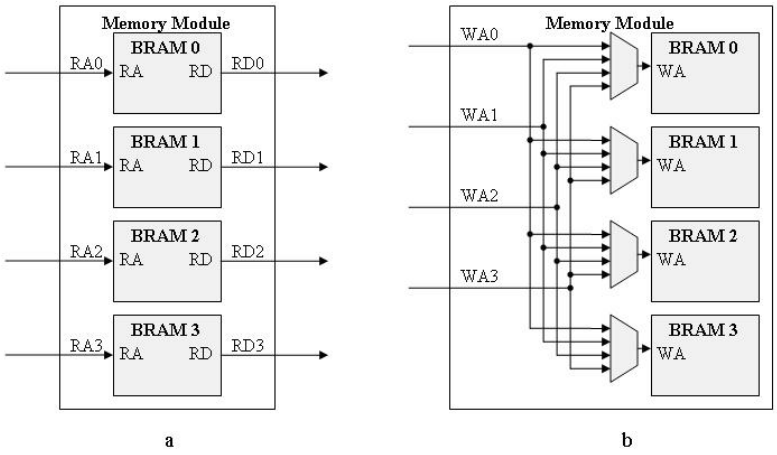


Fig. 3.3: Multiport memories using BRAMs. In (a), four read ports use four BRAMs. In (b), four write ports use four BRAMs, but four clock cycles are needed to allow a write from each port.

controller coordinates the sharing of data between processing stages, tracking the memory blocks associated with each stage. This coordination is done by managing the multiplexers that form the interface between memory banks and processing stages. The multiplexing allows for read and write access between any processing stage and any memory block. The connections are shown in Fig. 3.4 for read operations and Fig. 3.5 for write operations. Each

memory module has four read ports, to permit exploitation of parallelism in the Evaluate stage, and one write port. Read ports consist of address and data lines, while write ports consist of address, data, and write-enable lines. Upon completion of a pipeline period,

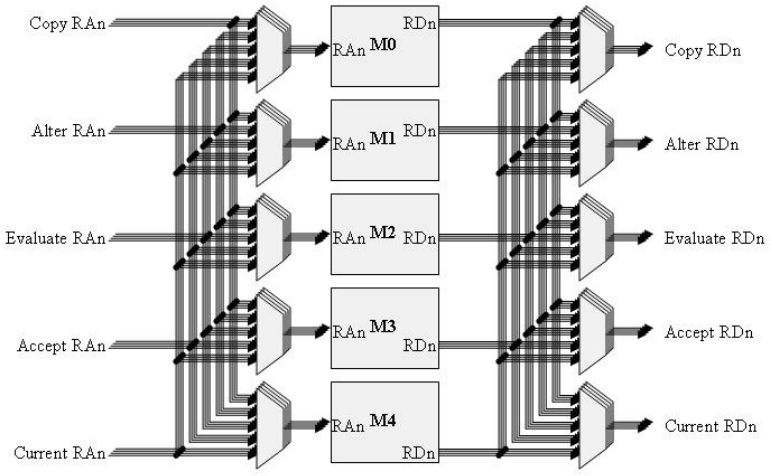


Fig. 3.4: Multiplexed connections between processing stages and memory modules for read accesses. Each memory block has four read ports.

the main controller reassigns the memory blocks to the different stages according to the rules shown in Fig. 3.6. Two different sets of rules are needed, depending on whether the solution in the Accept stage is rejected (Fig. 3.6a) or accepted as the new current solution (Fig. 3.6b). The main controller performs global synchronization of pipeline stages. When each stage completes execution it sends a done signal to the main controller. Once all stages have completed, the main controller issues a step signal to each stage, indicating that they can proceed to the next step. The main controller also monitors the temperature and halts the system when execution is complete.

3.2 The Copy Stage

Since the number of events in the solution is known to be v, the contents of the solution in the current solution memory bank are copied, word by word, into the memory bank currently associated with the Copy stage. The latency of this stage (t_c) is defined by

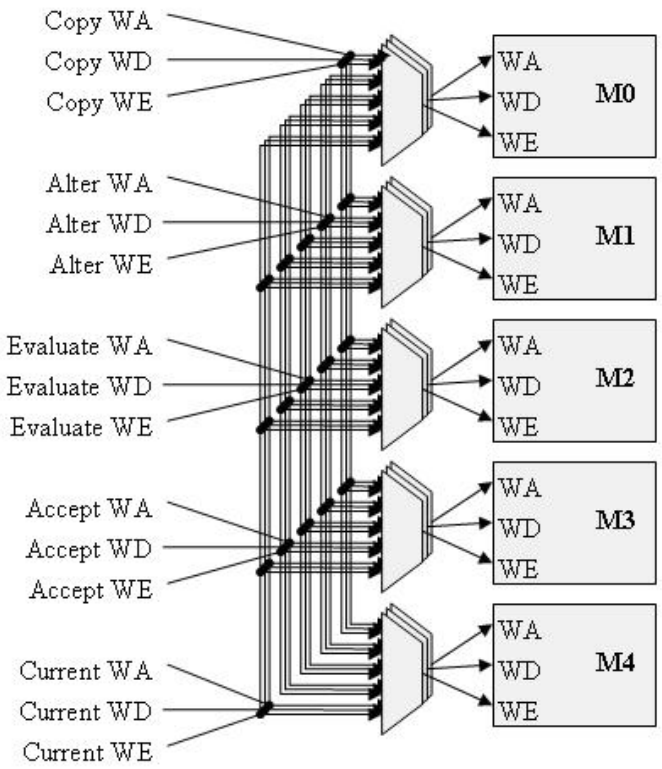


Fig. 3.5: Multiplexed connections between processing stages and memory modules for write accesses. Each memory block has one write port.

a

Copy Mem	→	**Alter Mem**
Alter Mem	→	**Eval Mem**
Eval Mem	→	**Accept Mem**
Accept Mem	→	**Copy Mem**
Current Mem	→	**Current Mem**

b

Copy Mem	→	**Alter Mem**
Alter Mem	→	**Eval Mem**
Eval Mem	→	**Accept Mem**
Accept Mem	→	**Current Mem**
Current Mem	→	**Copy Mem**

Fig. 3.6: Method for passing memory block pointers between processing stages when (a) the solution in the Accept stage is NOT accepted and (b) the Accept stage solution is accepted.

Eq. (3.1) as:

$$t_C = v + 1 \tag{3.1}$$

The step signal comes from the main controller to start the Copy stage. A counter generates addresses and produces a done signal when all data has been copied. As this stage is basically a counter, scalability is simply determining how wide the counter need be to count to v.

The Copy stage could be accelerated by either creating additional read and write ports or by widening the existing ports to allow faster data transfer. Fig. 3.3b and the associated discussion have already eliminated the possibility of creating additional ports, but the existing ports can be widened while incurring a substantial cost in BRAM usage. Fig. 3.7 depicts a memory system with a double-wide read (Fig. 3.7a) and write port (Fig. 3.7b) in addition to standard sized read and write ports. In Fig. 3.7a, a memory bank from Fig. 3.3 is divided into two BRAMs, where even-addressed words are stored in L and odd-addressed words in H. The low bit of the address becomes a select bit on the output multiplexer. For wide addresses, the outputs of both BRAMs are concatenated. In Fig. 3.7b, data is written to either the H BRAM or the L BRAM, depending on the address (odd or even, respectively). In case of wide writes, data from the wide bus is split and written into both BRAMs. The box marked E controls the memory write-enable lines accordingly. Implementing such an acceleration increases BRAM usage by the same factor as the number of words accessed in a wide access. In Fig. 3.7, wide accesses are twice the width of regular accesses, and thus use twice the number of BRAMs. Circuits can also be implemented with four-word or eight-word wide accesses, with proportional increases in BRAM usage. Additionally, introducing wide ports requires additional multiplexers and control units not needed in the simple design. Because of the pipelined nature of the AMPS accelerator circuit, the Copy stage need only be accelerated when it is the slowest stage. Because Evaluate is generally the most complex stage, the simple Copy model is generally sufficient.

3.3 The Alter Stage

The second stage in the iterative repair pipeline is the Alter stage. One event is selected

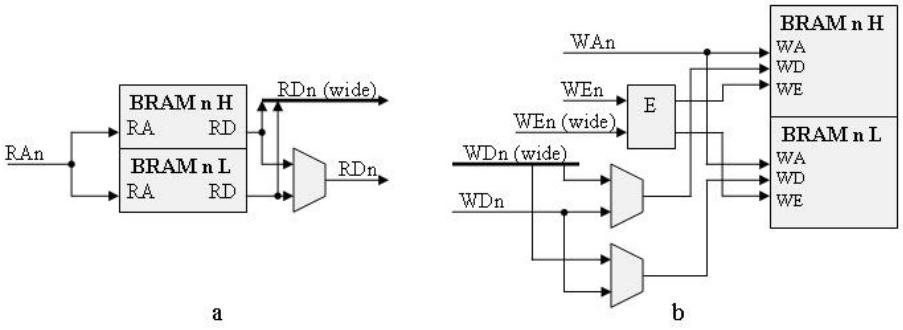


Fig. 3.7: Memory circuit that allows for double-wide data transfers in addition to word-sized accesses.

at random from the solution string. The start time of this event is changed to a random time that falls between zero and the maximum latency. The C code for this function is as follows:

```
i = rand() % MAX_EVENTS;
j = rand() % MAX_LATENCY;
sched[i] = j;
```

The hardware implementation of this stage, shown in Fig. 3.8, could be accelerated by introducing an additional random number generator and an additional divider, allowing for maximum concurrency. This additional hardware is not necessary however, as a 19-cycle integer divider allows this stage to terminate in 21 clock cycles, as shown in Eq. (3.2), regardless of the size of the solution string.

$$t_{AL} = 21 \tag{3.2}$$

The Alter controller is based on a counter that starts when the step signal is received from the Main Controller, control logic to enable register writing, and a done signal. This stage is made scalable by changing the constants that represent maximum events and maximum latency. Additionally, addresses must be wide enough to address all data locations.

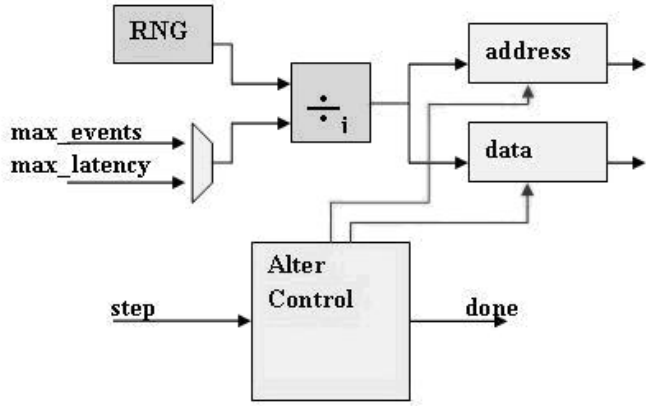


Fig. 3.8: The Alter stage.

3.4 The Evaluate Stage

The Evaluate stage is the most complex of all the pipeline stages in the iterative repair architecture. Its job is to compute a numerical score for a potential solution. The score of a solution to this particular iterative repair problem consists of three components. The first component is the total time steps consumed by the schedule, promoting shorter schedules. The second component is the number of times a resource is over-booked on a given clock cycle. The third component is the sum of the magnitudes of all dependency violations, which occur when event b depends upon the results of event a, but event b is scheduled before event a. These three partial scores are summed to produce the solution score.

Each of the three evaluation components described above is implemented as an individual pipelined sub-stage. Because the three components of the score can be computed independently and combined at the end, all three sub-stages can run in parallel, thus saving substantial clock cycles.

The first sub-stage, termed the Dependency Graph Violation sub-stage, or DGV, is shown in Fig. 3.9. The original C code from which this sub-module is derived is shown here:

```
for (i=0; i<MAX_EDGES; i++)
  if (next[source[i]] >= next[dest[i]])
      conflicts = conflicts + (next[source[i]] - next[dest[i]]) + 1;
```

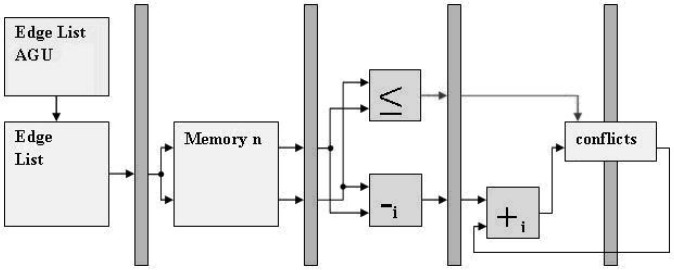


Fig. 3.9: The dependency graph violation substage (DGV).

The DGV sub-stage is itself a four-stage pipeline. In the first and second stages, edge source and destination lists are used to index the solution memory and determine when parent/child pairs of events are scheduled. Edges are kept in a list of source/destination pairs. The execution time of the DGV sub-stage is directly proportional to the length of the dependency list. The third and fourth stages determine the magnitude of the penalty, if any, to be incurred because a child event is scheduled to begin before a parent event terminates. DGV execution time is formalized in Eq. (3.3), where three cycles are added to account for initial pipeline filling.

$$t_{DGV} = E + 3 \tag{3.3}$$

The second sub-stage, shown in Fig. 3.10, is the Total Schedule Length sub-stage (TSL). It computes the total length of the schedule from beginning to end. Events are read one-by-one, updating the earliest and latest times seen so far. Upon conclusion, the difference between the earliest and latest times is the schedule length. The C code for this process is shown here:

```
for (i=0; i<MAX_EVENTS; i++)
{
   if (sched[i]<start) start = sched[i];
   if (sched[i]>stop) stop = sched[i];
}
conflicts = stop - start;
```

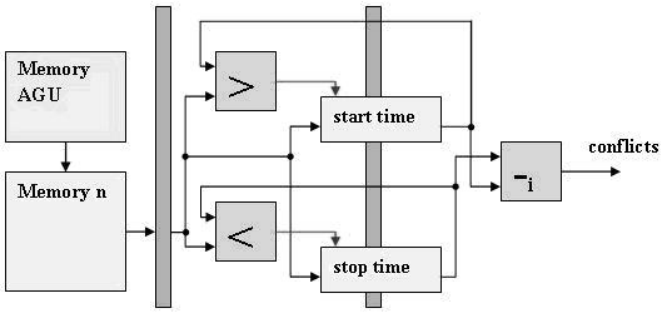


Fig. 3.10: The total schedule length sub-stage (TSL).

Execution time is proportional to the number of events and is formalized in Eq. (3.4), with one cycle added for initial pipeline filling.

$$t_{TSL} = v + 1 \tag{3.4}$$

The third sub-stage internal to the Evaluate stage is the Resource Over-Utilization stage (RO). This sub-stage, depicted in Fig. 3.11, is responsible for checking for resource over-scheduling on every resource for every time step. A two-dimensional matrix keeps track of the resource utilization of every resource for every time step. This matrix is first cleared, and then updated by going through the events one by one and determining when each is scheduled and what resource each uses. When a scheduled event causes a resource over-utilization, a counter is incremented. The C code for this process is shown here:

```
for (i=0; i<MAX_RESOURCE_TYPES; i++)
{
   for (j=0; j<MAX_LATENCY; j++)
   {
      t_matrix[j][i] = 0;
   }
}
for (i=0; i<MAX_EVENTS; i++)
{
   t_matrix[next[i]][resource_usage[i]]++;
   if (t_matrix[next[i]][resource_usage[i]] >
      resources[resource_usage[i]])
   {
```

```
            conflicts++;
        }
    }
```

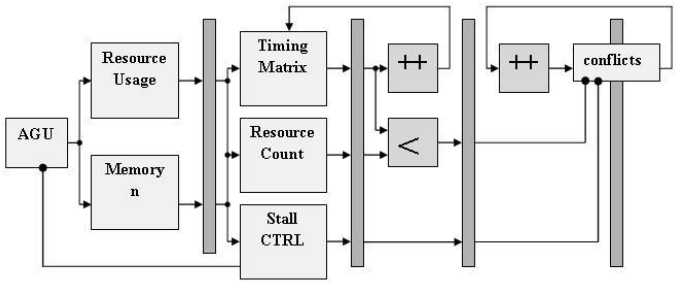


Fig. 3.11: The resource over-utilization sub stage (RO).

Two timing matrices are actually present in the circuit. On a given iteration, one is used while the other is cleared, removing the additional time consumed by clearing from the stage latency. Because RO is a four-stage pipeline, stalls must be inserted to prevent read/write conflicts in the timing matrix. If two subsequent events read from the main memory access the same location in the timing matrix, the read operation for the second event stalls for one cycle to allow the write operation for the first event to complete. The best and worst case execution time for RO are computed using Eq. (3.5) and Eq. (3.6), respectively. The best case execution time is the slowest performer between the RO pipeline when no stalls occur and the clearing of the timing matrix not in use. The worst case execution time is the slowest performer between the RO pipeline when every possible stall occurs and the clearing of the timing matrix not in use.

$$t_{ROB} = max(v + 3, t_r L_{max} + 1) \tag{3.5}$$

$$t_{ROW} = max(2v + 2, t_r L_{max} + 1) \tag{3.6}$$

TSL, DGV, and RO all have done signals. When all three have completed their tasks, the three penalty values are weighted and combined to give the total score for the given schedule. This score is stored in the associated main memory bank. The Evaluate stage is made scalable by changing the parameters representing events, edges, latency, resource

types, and number of resources. The performance of the Evaluate stage is bounded by two equations, computing the best Eq. (3.7) and worst Eq. (3.8) execution times.

$$t_{EB} = max(t_{DGV}, t_{TSL}, t_{ROB}) \tag{3.7}$$

$$t_{EW} = max(t_{DGV}, t_{TSL}, t_{ROW}) \tag{3.8}$$

While the Evaluate stage has been parallelized at a high level into RO, TSL, and DGV sub-stages, one might advocate increased low-level parallelism as well through loop unrolling. For example, an attempt might be made to speed up the RO sub-stage by replicating the circuit shown in Fig. 3.11, mapping half of the iterations to each replication, and combining results. Unfortunately, this would necessitate sharing the timing matrix across all unrolls and requiring simultaneous data write operations. As discussed previously, increasing the number of write ports above the one permitted by VHDL mapped by Xilinx ISE onto BRAMs is not possible without serious degradation to performance, this approach is not advocated.

3.5 The Accept Stage and the Adjust Temperature Module

The Accept stage determines whether to accept the next solution as the new current solution. If the next solution is better than the current solution, the next solution is accepted unconditionally. A solution that is worse than the current solution can also be accepted with a computed probability, defined in Eq. (2.2). The code for this process is shown below.

```
delta_e = cur_value  next_value;
p = exp(((float)delta_e)/temperature);
if ((rand() / (float) RAND_MAX) < p)
{
   for (i=0; i<MAX_EVENTS; i++)
      schedule[i] = next_schedule[i];
   cur_value = next_value;
}
```

An architecture that supports this computation is shown in Fig. 3.12. This stage mixes floating-point and integer arithmetic, thus necessitating the integer-to-float conversion module shown. The *cur_value* and *next_val* variables are integers, while *temperature* and $RAND\_MAX$ are 32-bit floating point numbers. The current score and the next score are read from their respective memory banks. The temperature is provided by the Adjust Temperature stage. The random number generator and divider produce a number between zero and one that is compared against the acceptance probability (p) to determine whether or not the new solution should be accepted. The Accept stage needs no modification to

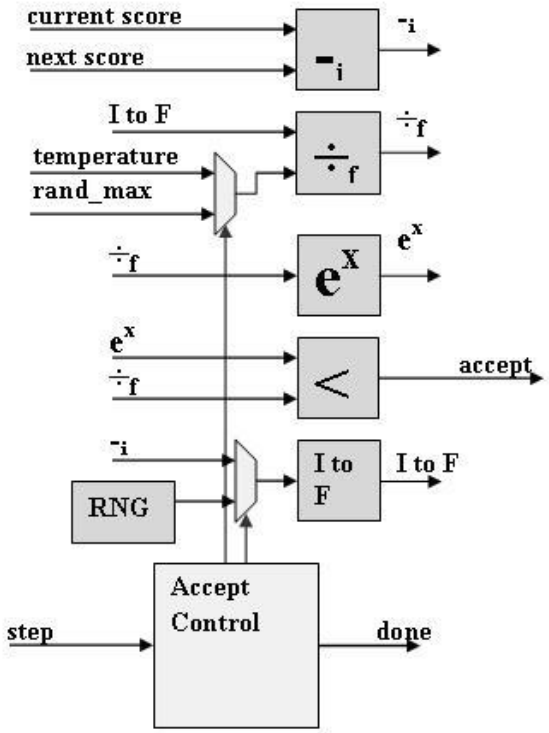


Fig. 3.12: The Accept stage.

become scalable. The input is always two 16-bit integers and the output is a single bit indicating whether the solution should be accepted. With an integer adder taking one cycle, a floating point divider taking 19 cycles, the exponential unit (implemented as a look-up table) taking one cycle, integer-to-floating point conversion taking six cycles, and the random number generator taking one cycle, the latency of this stage is a constant,

defined in Eq. (3.9).

$$t_{AC} = 54 \tag{3.9}$$

The Adjust Temperature module is a simple but critical module. The temperature is used to compute the probability of acceptance in the Accept stage and by the main controller to determine when execution is complete. There are many options for implementing a cooling schedule for a simulated annealing problem. In this case a popular geometric cooling rate is employed, although this could easily be replaced with a different function deemed more appropriate for a specific application. The architecture for the Adjust Temperature module is shown in Fig. 3.13. The current temperature is stored in a register. When the step signal is received, the temperature is multiplied by the constant cooling rate, generally a floating point value slightly less than one. This cooling rate allows the temperature to decrease slowly and geometrically, permitting the algorithm hundreds or thousands of iterations to discover better solutions. As this module is not directly related to the solutions

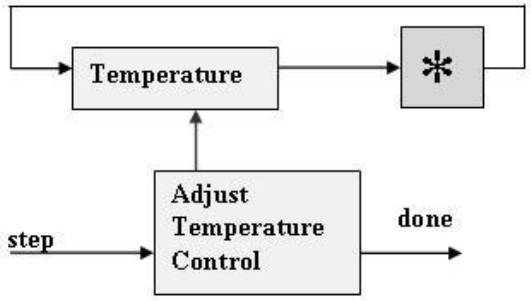


Fig. 3.13: The Adjust Temperature module.

in progress, no changes are needed to make this module scalable. The module performance is characterized by Eq. (3.10).

$$t_{AT} = 12 \tag{3.10}$$

3.6 Summary of the Architecture and System Performance Characterization

A pair of equations that describes the best (t_B) and worst (t_W) performance in time of the architecture is shown in Eq. (3.11) and Eq. (3.12), where the number of stalls in the

RO sub-stage of the Evaluate stage cause the difference between t_{EB} (best-case evaluate time) and t_{EW} (worst-case evaluate time).

$$t_B = max(t_C, t_{AL}, t_{EB}, t_{AC}, t_{AT}) \tag{3.11}$$

$$t_W = max(t_C, t_{AL}, t_{EW}, t_{AC}, t_{AT}) \tag{3.12}$$

Because the architecture is a high-level pipeline, execution can only proceed as fast as the latency of the slowest stage. Fig. 3.14 shows best-case and Fig. 3.15 shows worst-case possible performances of the architecture for problems consisting of a 32-cycle maximum schedule latency, four resource types, a maximum of five resources of each type, up to 1000 events, and up to 1500 edges. In addition to predicting processor latency, equations can

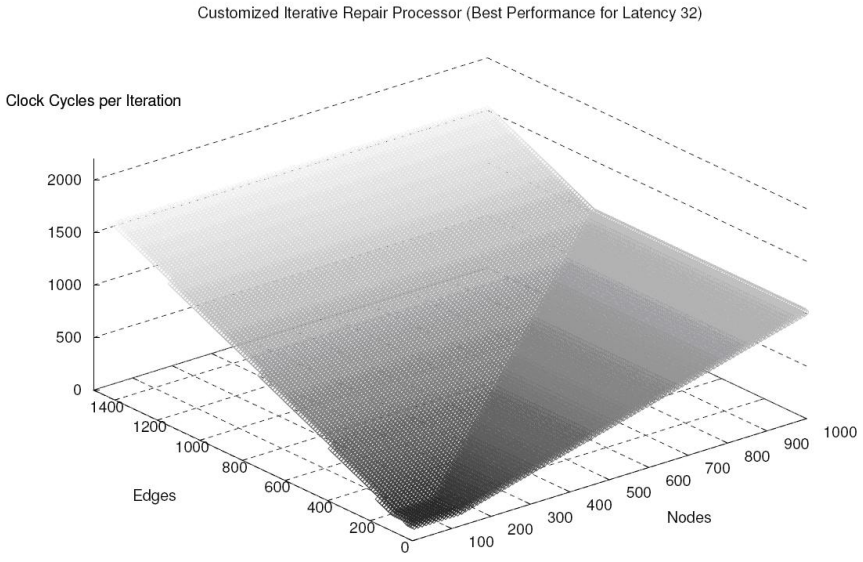


Fig. 3.14: Best-case AMPS accelerator performance.

also be derived to describe FPGA resource utilization. The physical sizes of the processing stages change very little from problem to problem, and are not interesting to quantify. The amount of memory usage, however, can vary substantially. Total memory usage is the sum of the usage of five different memory modules and its estimation is computed as follows. First, the main memory always consists of five memory banks, each with four read ports

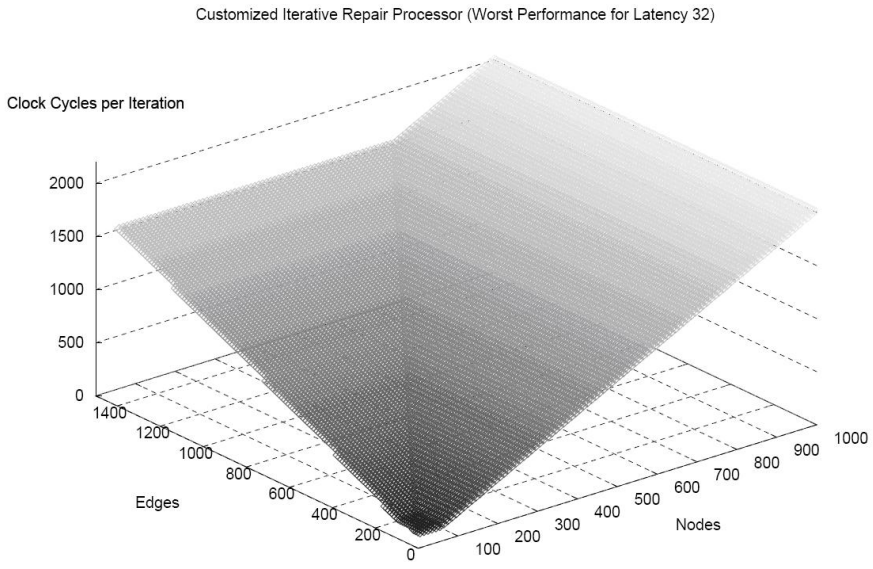


Fig. 3.15: Worst-case AMPS accelerator performance.

and a single write port. Data width is 16 bits. BRAM memory usage for main memory is described by Eq. (3.13).

$$B_M = 20\lceil \frac{16v+1}{b} \rceil \tag{3.13}$$

The timing matrix in the RO sub-stage of the Evaluate stage also uses BRAMs. Remember that two timing matrices exist, operating together as a ping-pong buffer. RO BRAM usage is described by Eq. (3.14).

$$B_T = 2\lceil \frac{16t_r L_{max}}{b} \rceil \tag{3.14}$$

The edge list employed by the DGV sub-stage also uses BRAMs. There is an entry in the list for every edge, with every edge consisting of a source and a destination event identifier. Thus, each entry must be wide enough to hold the twice the length of the largest possible event identifier. DGV BRAM usage is described in Eq. (3.15).

$$B_T = 2\lceil \frac{2E\lceil log_2 v \rceil}{b} \rceil \tag{3.15}$$

These equations to define processor performance in speed and area are compared with measured results of various problems later in this chapter.

3.7 Differences between Software and Hardware Implementations

At this stage a discussion on a significant difference between the operation of the IR algorithm in hardware versus software is presented. Because of the four-stage pipeline present in the AMPS accelerator circuit, there is a difference between the simulated annealing software and the custom hardware designs. In software, simulated annealing keeps track of only two solutions at any time the current solution and a potential next solution. In the pipelined custom hardware design, however, what should be done with the solutions in the Copy stage, the Alter stage and the Evaluate stage when a new solution is accepted by the Accept stage? In the sequential software implementation, this issue does not exist, as there is no high-level pipeline with multiple solutions in progress to worry about. This problem can be solved in the hardware implementation in one of two ways, either (1) flush the pipeline, which is consistent with the software version of simulated annealing, or (2) simply ignore the issue. In this architecture, we opted for solution two because of its simplicity. Flushing the pipeline would require additional circuitry to unconditionally reject outdated solutions. However, even though the solutions in the Copy, Alter, and Evaluate stages were created from a solution that is no longer the current solution, they are still valid potential solutions and can be treated as such. This saves the additional circuitry and delays needed to flush the pipeline.

3.8 Enhancements through PDR and TMR

3.8.1 PDR applied to the Evaluate sub-system

The AMPS accelerator circuit can schedule different sets of events and dependencies with different sets of constraints like available resources, available time, etc. The results of the three Evaluate sub-stages (RO, TSL, and DGV) may need to be weighted differently from graph to graph in order to reflect these changing constraints. Partial Dynamic Reconfiguration (PDR) can be employed to replace only the Evaluate module, saving substantial reconfiguration time over what would be required to reconfigure the entire FPGA. A dynamic reconfigurable Evaluate stage has been tested targeting an ML402 Development

board with a XilinxVirtex 4 SX35 FPGA onboard using Early Access Partial Reconfiguration (EAPR) [77], a partial reconfiguration methodology from Xilinx.

A block diagram of the PDR system setup is shown in Fig. 3.16. The Evaluate stage is separated from the main processor and designed as a top module and placed in a Partially Reconfigurable (PR) region to permit reconfiguration. The communication between the evaluate module and the main processor is realized using slice based bus macros. Each bus macro has the capability of communicating eight bits of data. OPB HWICAP, which is a wrapper for the Internal Configuration Access Port (ICAP), is used to perform reconfiguration. The interface to the outside world is done using the OPB UART. The initial configuration file and the partial bit streams are stored in a compact flash card which is controlled by the SystemACE chip on the ML402 evaluation board. An OPB Timer is also included in the system which can be used to measure the number of clock cycles needed to execute any routine. All the peripherals are controlled by an on-FPGA MicroBlaze soft core processor and all communication between the peripherals is done through the on-chip peripheral bus (OPB). The PR region can hold one of two reconfigurable modules, *evaluate_1* and *evaluate_2*. *Evaluate_1* is as described previously. *Evaluate_2* has the same architecture, with different weightings of two of the three penalties: s_{RO} is multiplied by two and s_{TSL} is divided by two. In other words, more importance is given to avoiding resource over-utilization and less to total schedule length. As discussed previously, different weightings can apply to different situations, dictated by mission parameters and environmental changes.

3.8.2 TMR applied to the entire AMPS accelerator circuit

Space is an extremely noisy environment. High-energy particles can impact an electronic circuit, causing errors in logic states. For example, a particle can impact a transistor and temporarily switch a logic one to a logic zero or a logic zero to a logic one. This transient error is called a Single Event Upset (SEU) and the occurrence of such faults must be handled by the circuit. For FPGAs, SEUs can also occur in the configuration memory. Errors in the configuration memory are repaired by periodically reloading the FPGA with

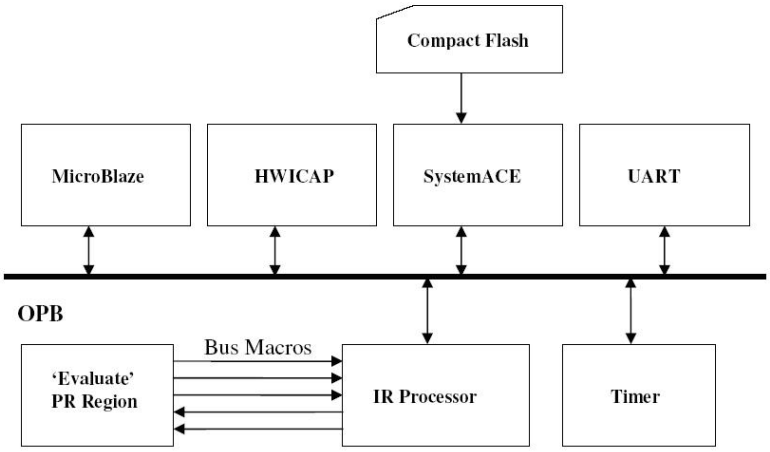


Fig. 3.16: System block diagram for partial reconfiguration of Evaluate module.

the original fault-free configurations. A standard method for SEU protection is Triple Modular Redundancy (TMR) [78]. In TMR, three copies of a circuit are implemented and the results are passed through a majority voter to ensure correct results. TMRTool, provided by Xilinx, automatically converts designs to a TMR state and replaces any shift registers used [76] to remove components that use configuration frames for memory. This prevents the accumulation of SEUs in the configuration memory by reloading the frames as discussed earlier. TMRTool has been shown to provide robust protection for SRAM FPGAs against SEUs [79–81]. This tool is employed to generate a fault-tolerant version of the AMPS accelerator circuit.

Xilinx also offers a line of radiation-hardened FPGAs for military and aerospace applications (Virtex II-Q and Virtex II-Q Pro series, QV4 etc.). These devices have very similar performances to their non-rad-hard counterparts and provide protection against Total Ionization Dose. Combining TMR with a radiation-hardened part will provide protection from SEUs and accumulated radiation.

3.9 Performance of the AMPS Accelerator Circuit

The parameterized nature of the AMPS accelerator circuit allows a customized version to be instantiated on an FPGA for a wide range of problem sizes. The Xilinx TMR tool

discussed in Chapter 2 was also employed to verify that fault protection is indeed possible for these circuits. Fig. 3.17 to Fig. 3.20 show resource usage on the FPGA for ten randomly-generated example event dependency graphs. The number of events varies from 41 to 981, the number of edges varies from 51 to 1,317, and the maximum latency allowed for the schedule varies from 32 to 256. Resource utilization is divided into lookup tables (LUTs) (Fig. 3.17), flip-flops (Fig. 3.18), embedded block RAM modules (BRAMs) (Fig. 3.19), and embedded DSP48 signal processing units (Fig. 3.20) for both simple and TMR-protected circuits. Power usage, estimated using the Xilinx XPower tool, is also provided for both circuit types in Fig. 3.21 and ranges from 569 mW to 892 mW. This is significantly lower than the PowerPC 750, with quoted power usage of less than 5 W [82]. It can be seen

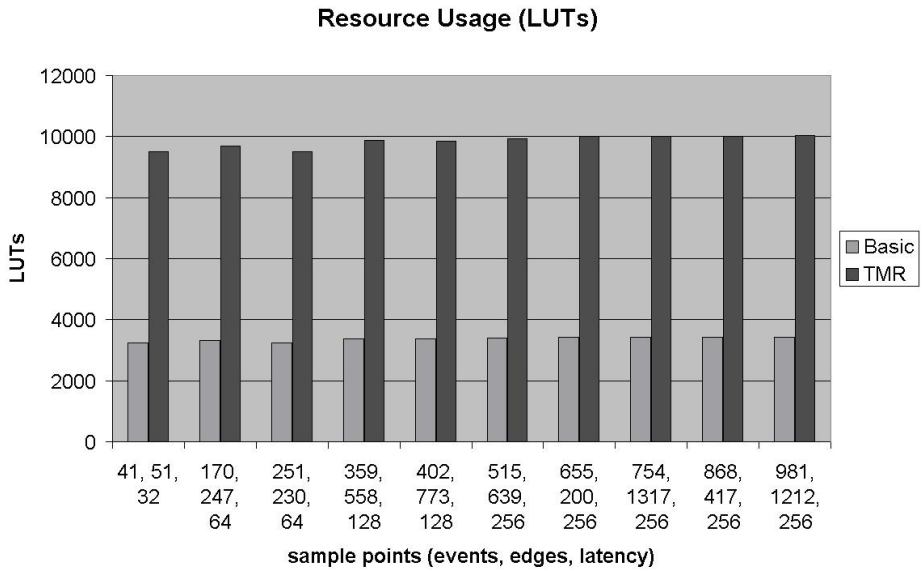


Fig. 3.17: Resource usage (LUTs) for 10 example problems.

that resource usage increases by at least a factor of three across all resource types for all designs that use TMR. This is to be expected, as the base circuit is triplicated, shift-registers extracted, and voters added. Power usage increases by about 50 percent on average for a TMR design, as power consists of both quiescent and dynamic components. The target device for these experiments was the Xilinx V4SX35, which consists of 30,720 LUTs, 30,720 flip-flops, 192 DSP48 units, and 192 BRAM blocks. All example designs easily fit on this chip.

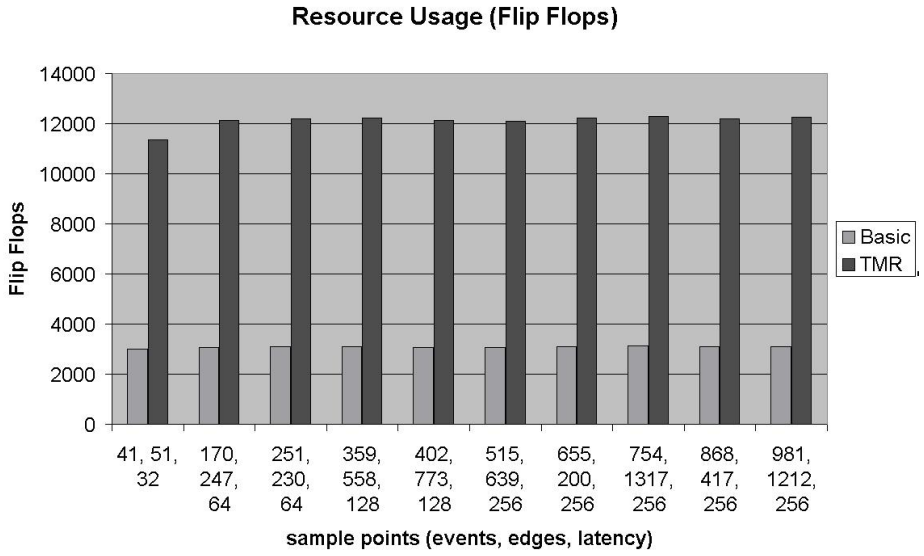


Fig. 3.18: Resource usage (flip-flops) for 10 example problems.

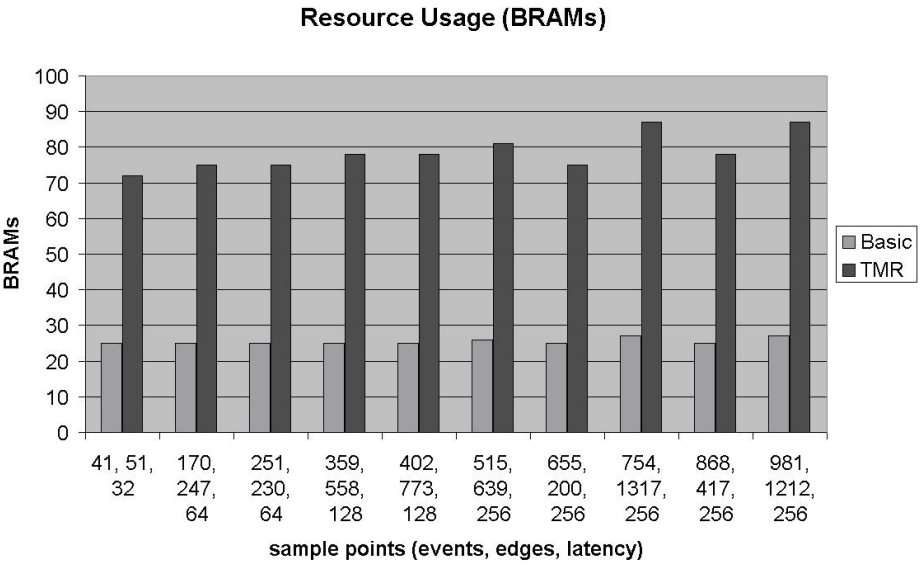


Fig. 3.19: Resource usage (BRAMs) for 10 example problems.

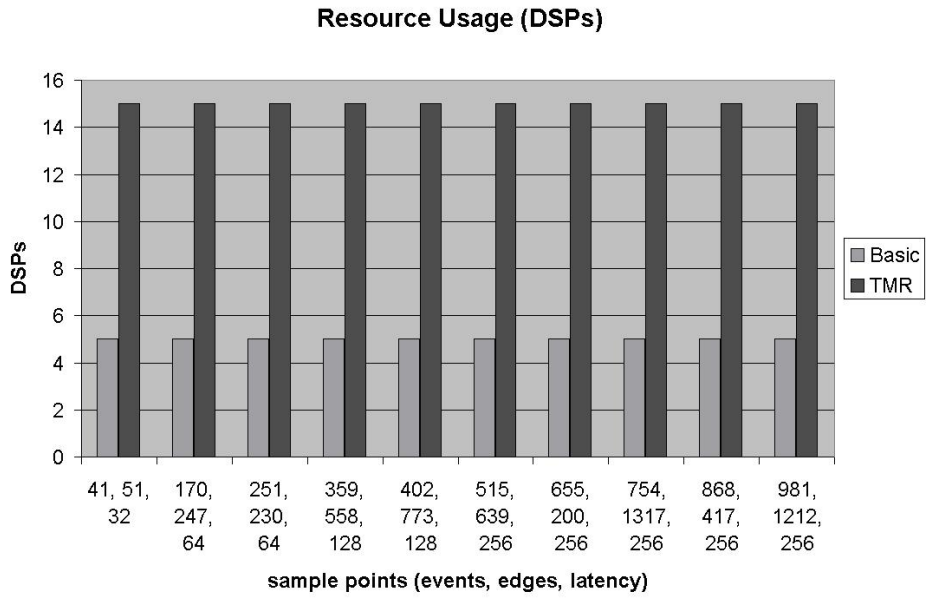


Fig. 3.20: Resource usage (DSP48s) for 10 example problems.

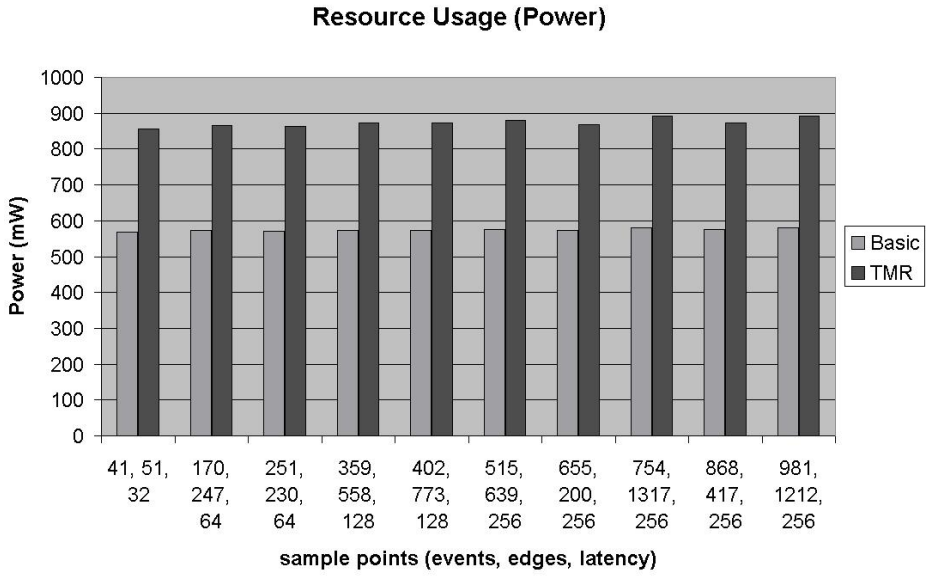


Fig. 3.21: Power usage for 10 example problems.

Table 3.2: Reconfiguration Time of Evaluate and AMPS Accelerator Circuit

Reconfigurable Module	No. of bits in the Bitstream	Time (in ms) SystemACE	Time (in ms) On-chip BRAM
Evaluate	498,040	220	19.9
AMPS Accelerator Circuit	2,245,008	830	N/A

Table 3.3: Architecture Resource Usage of Fault Injected Circuits

LUTs			Flip-Flops			BRAMs			DSP48s		
Basic	TMR	Ratio	Basic	TMR	Ratio	Basic	TMR	Ratio	Basic	TMR	Ratio
3,057	9,888	3.23	3,837	11,589	3.02	26	78	3.00	5	15	3.00

The results of the partially and dynamically reconfigurable AMPS accelerator circuit are shown in Table 3.2. The results of the partially and dynamically reconfigurable AMPS accelerator circuit are shown in Table 3.2. The time to reconfigure *evaluate_1* and *evaluate_2* is same as the size of the bit-stream is the same for both modules which lie in the same PR region. The time to reconfigure the entire AMPS accelerator circuit is four times more than just reconfiguring the evaluate module. The final column in Table 3.2 represents the time to reconfigure when the bit-stream is stored in on-chip BRAMs. Storing bit-streams in on-chip BRAMs rather than off-chip on a compact flash card achieves a speedup of 11 times. The bit-stream of the entire AMPS accelerator circuit is too big to fit in the on-chip BRAMs.

Table 3.3 and Table 3.4 present AMPS accelerator circuit fault protection results from the fault injector. A 100-node, 99-edge graph is taken as an example. The results show that TMR AMPS accelerator circuit greatly reduces the number of sensitive bits when compared to the unprotected AMPS accelerator circuit by a factor of 1800 times. The main restriction with using TMR is the additional area cost required in order to achieve such high protection. Fig. 3.22 compares the execution of the custom architectures running on the

Table 3.4: Sensitivity of Fault Injected Circuits

Number of Configuration Bits in Partial Configuration Area			Number of Sensitive Faults			Percent of Sensitive Bits in AMPS Accelerator Circuit		
Basic	TMR	Ratio	Basic	TMR	Ratio	Basic	TMR	Ratio
4,153,860	8,353,176	2.01	126,048	68	0.0005	3.03	0.0008	0.0003

FPGA versus identical algorithms running on a PowerPC 750 processor with floating point coprocessor. Notice that the y-axis in Fig. 3.22 is a logarithmic axis. As the BAE Systems RAD750 processor is extremely expensive (hence not affordable by an academic institution), a cycle-accurate PowerPC 750 emulator from Virtutech was employed to obtain timing data. For space applications, power usage is a significant concern. Since power usage is directly proportional to clock speed, clock speeds are generally confined to 100 MHz or 150 MHz for most applications. Therefore, the comparisons in Fig. 3.22 assume a frequency of 100 MHz for all devices. Both the PowerPC and the custom circuits could be clocked faster, but comparative results would be identical. Over the range of test cases, it can be seen that the custom circuits out-perform the PowerPC by roughly 30 to 90 times. Fig. 3.22 also includes best and worst-case execution times for each custom circuit. Finally, the results of the

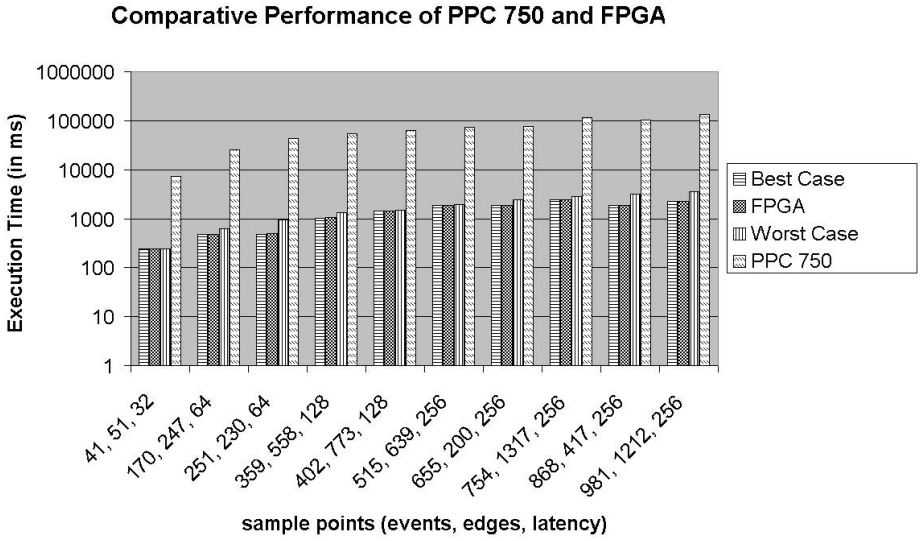


Fig. 3.22: Execution times for 10 example problems on both custom hardware and PowerPC 750.

hardware circuit are compared with those of the algorithm running in software. Fig. 3.23 compares a 100 event, 99 edge graph scheduled in both hardware and software. In both cases, the experiment was repeated 20 times and the results averaged. It can be seen from Fig. 3.23 that both methods follow the traditional simulated annealing curve. However, the hardware version actually performs better initially, with the software version catching up

towards the end of execution. The difference in functionality of the hardware version is due to the lack of flushing in the pipelined architecture when a new solution is accepted.

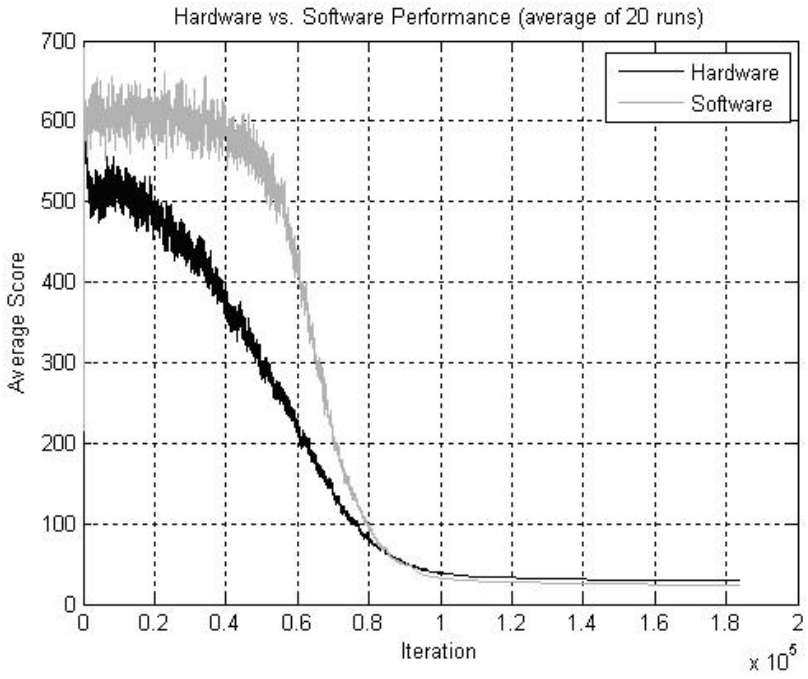


Fig. 3.23: Hardware vs. software results comparison.

Chapter 4

Architecture Derivation Methodology

This chapter describes a novel approach to providing a path from C code specification to efficient hardware implementation. Fig. 4.1 shows design steps that could be followed both by software engineers to produce executable code and hardware engineers to produce custom hardware. In both cases, the starting point is a conceptual algorithm, usually expressed in written English, which describes the general set of operations that needs to be performed. This written description can be formalized as a control data flow graph (CDFG) that details the operations that need to be performed using building blocks such as add, subtract, multiply, load, store, etc. This is step does not usually occur in software or hardware design, but an implementation-independent CDFG can be produced for any algorithm.

At this point, Fig. 4.1 shows branching paths for software and hardware design. The software path calls for sequentialization of the CDFG into a pseudocode representation. Because traditional software is targeted for execution on a sequential machine (instructions are executed in order, one at a time), an ordering must be applied to the CDFG. Some order is imposed by connections in the CDFG, but otherwise the ordering is arbitrary. Thousands of permutations of pseudocode may exist for a single CDFG. Once the pseudocode has been generated, the next step for a software engineer is to write the actual code in a chosen programming language such as C, C++, or Java. This code is generally sequential, following the same flow as the pseudocode. Compilers, assemblers, and linkers are then employed to produce the final executable file. On the other hand, a hardware designer takes the CDFG and breaks it into a hierarchy of modules. A hand-drawn or CAD-generated drawing of the hierarchy is generally developed at this point. A hardware description language (HDL), generally Verilog or VHDL, is chosen, and the hierarchical modules are described, simulated,

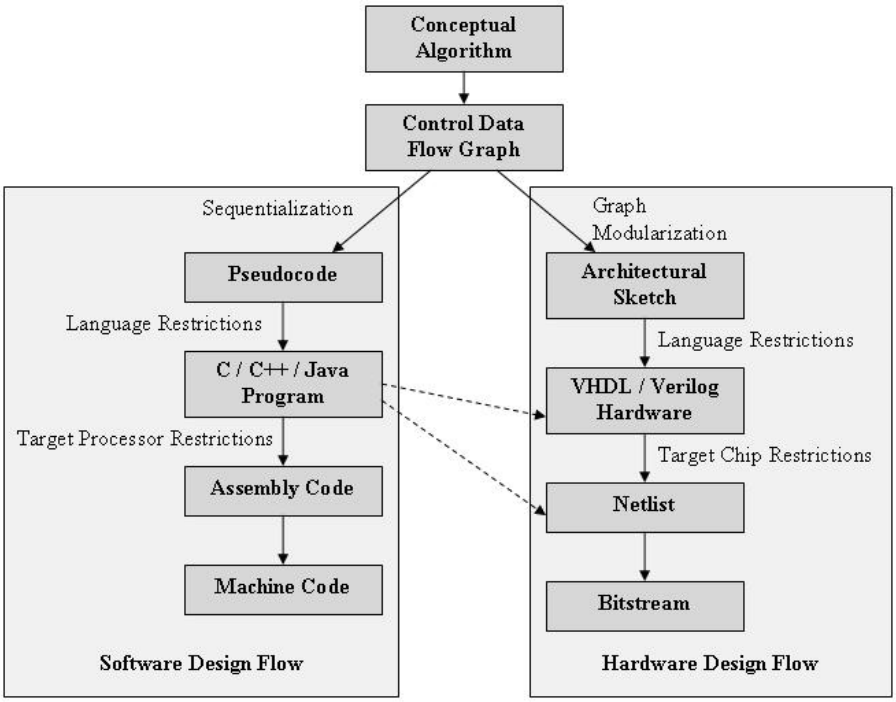


Fig. 4.1: Typical steps for hardware and software design.

and debugged. Synthesis and place-and-route tools are then employed to generate the final bit-stream that represents the verified circuit.

While most of the software-to-hardware tools described in chapter 2 are proprietary and underlying algorithms are not publicly available, these tools generally appear to follow an algorithm shown by the dotted lines in Fig. 4.1. High-level source code is translated directly into HDL or directly into a netlist format. While this technique has proved marginally successful, there are inherent weaknesses in this approach. Primarily, the flexibility of the software program is greatly reduced from that of the initial CDFG. The sequentialization and language-specific restrictions present in code impose unnecessary restrictions on hardware design. The ideal path for hardware derivation from software code would be to backtrack in Fig. 4.1 from software program to CDFG and then proceed in an automated manner down the hardware design path. A novel methodology, entitled SATH (Simulated Annealing to Hardware) is proposed which follows this algorithm.

In the previous chapter, a hardware architecture template was discussed for the simulated annealing architecture. This template is employed as part of the architecture derivation design flow. Fig. 4.2 shows a block diagram of the steps needed to translate high-level software code into a hardware representation. The remainder of this chapter will describe each step in Fig. 4.2 in detail.

4.1 Functional Block Partitioner

Given the simulated annealing pseudocode shown in Fig. 2.8 and the hardware architecture template shown in Fig. 3.2, the task of the Functional Block Paritioner (FBP) is to perform an initial mapping of software functions to hardware blocks. There is generally a one-to-one correspondence between software and hardware. In other words, a software function becomes a hardware block. Software functions such as copy, alter, evaluate, accept, and adjust temperature are mapped to equivalent hardware blocks.

4.2 Constant Extractor

Constants describing parameters of the simulated annealing algorithm are identified and extracted from the source code. A simulated annealing code contains parameters such as initial temperature, cooling rate, and cutoff level, as seen in the pseudocode in Fig. 2.8. These parameters are needed by the main hardware controller, the Accept stage, and the Adjust Temperature module shown in Fig. 3.2. The size of the solution array is also extracted for use by the Copy stage, the Alter stage, and the Evaluate Stage.

4.3 GCC

Once the source code has been partitioned, portions to be fed to the SAM stage need to be reduced to a control data flow graph. The first step is to use an off-the-shelf compiler like GNU GCC to produce a C-like three-address single-assignment code. An example of this conversion is shown in Fig. 4.3. This step is important for two reasons. First, three-address code removes any operation chaining within a single instruction. Any statement has at most three simple terms, with a maximum of two to the right of an equal sign in an assign

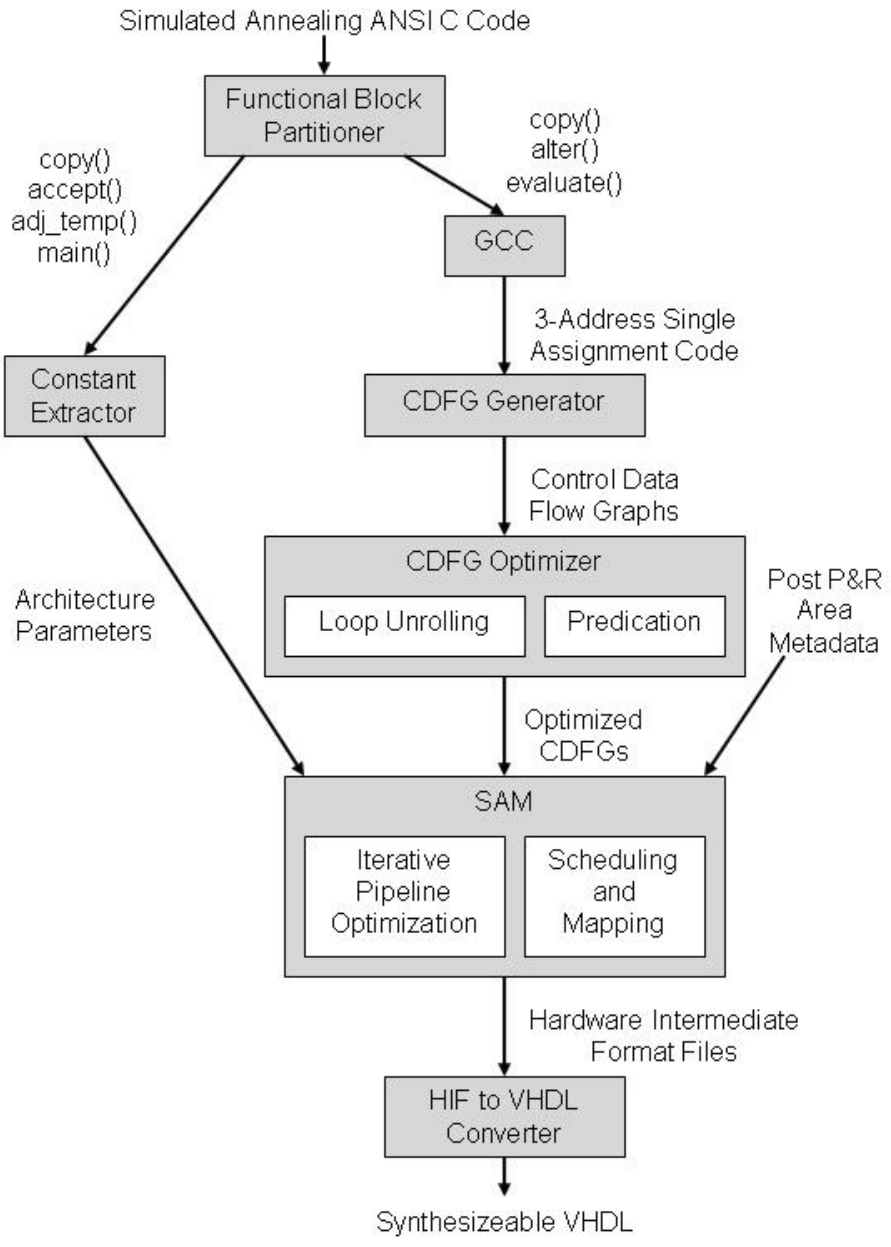


Fig. 4.2: Block-level diagram for translating simulated annealing C code to hardware.

statement. This makes the code much easier to interpret. Second, single-assignment code means each variable is found on the left of an assignment statement at most one time. This greatly simplifies the task of deriving data-flow patterns and determining variable liveness.

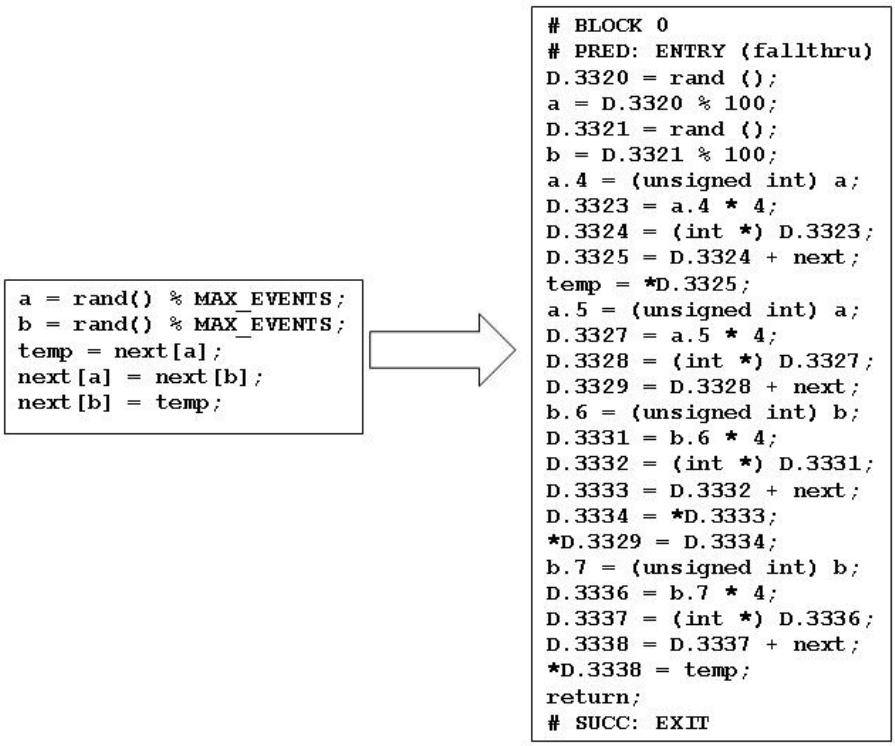


Fig. 4.3: Example of source C code and comparable 3-address single assignment code.

4.4 CDFG Generator and Optimizer

As described at the beginning of this chapter, the software code must be generalized to a CDFG, as the CDFG is the branching point between software and hardware design. To derive a CDFG, both arbitrary serialization of code and programming language-based restrictions must be removed. Fig. 4.4 shows an example of three-address code translated into a preliminary CDFG with serialization removed. The CDFG shown in Fig. 4.4 must then be further generalized by removing programming language-specific constructs. This includes loop unrolling and removal of base-plus-offset computations. Fig. 4.5 shows an

```
# BLOCK 0
# PRED: ENTRY (fallthru)
D.3320 = rand ();
a = D.3320 % 5;
D.3321 = rand ();
b = D.3321 % 5;
a.4 = (unsigned int) a;
D.3323 = a.4 * 4;
D.3324 = (int *) D.3323;
D.3325 = D.3324 + next;
temp = *D.3325;
a.5 = (unsigned int) a;
D.3327 = a.5 * 4;
D.3328 = (int *) D.3327;
D.3329 = D.3328 + next;
b.6 = (unsigned int) b;
D.3331 = b.6 * 4;
D.3332 = (int *) D.3331;
D.3333 = D.3332 + next;
D.3334 = *D.3333;
*D.3329 = D.3334;
b.7 = (unsigned int) b;
D.3336 = b.7 * 4;
D.3337 = (int *) D.3336;
D.3338 = D.3337 + next;
*D.3338 = temp;
return;
# SUCC: EXIT
```

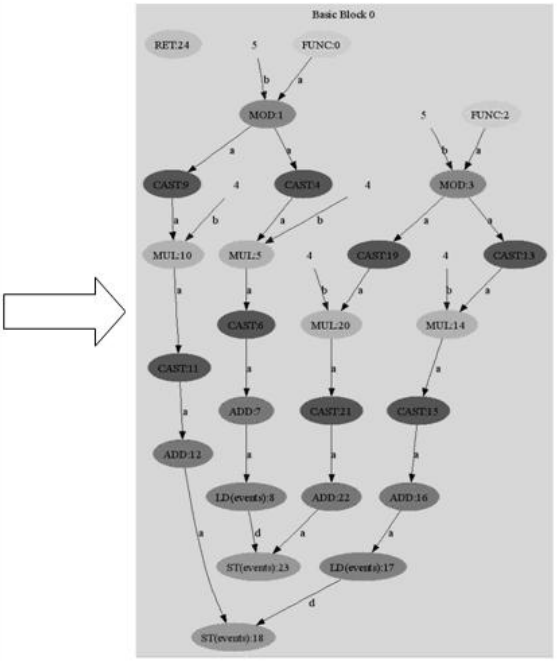


Fig. 4.4: Obtaining a CDFG from 3-address single assignment code.

example of this simplification. Specifically, the return node is removed and four distinct base-plus-offset computations are removed (identified as cast-multiply-cast-add chains). In addition to the modifications already described, three other changes are made to the CDFG to prepare it for processing by the SAM stage. First, loops are unrolled completely. The SAM algorithm only supports loops where the iteration count is a compile-time constant. Second, if-then-else constructs are redefined for predicative execution. Essentially, both true and false branches of an if-then-else construct are computed and the correct output is chosen through a selection operation. Third, constant expressions are simplified. When loop unrolling takes place, the loop iterator becomes a constant in each iteration. If both inputs to an arithmetic node are constants, that operation can itself be reduced to a constant. A more-complex example showing loop unrolling, if-statement conversion, and constant-expression simplification is shown in Fig. 4.6. The loop is unrolled four times. Notice that the non-optimized CDFG in Fig. 4.6 consists of six basic blocks, while the optimized version is a single block. All control flow has been removed from the graph, reducing the

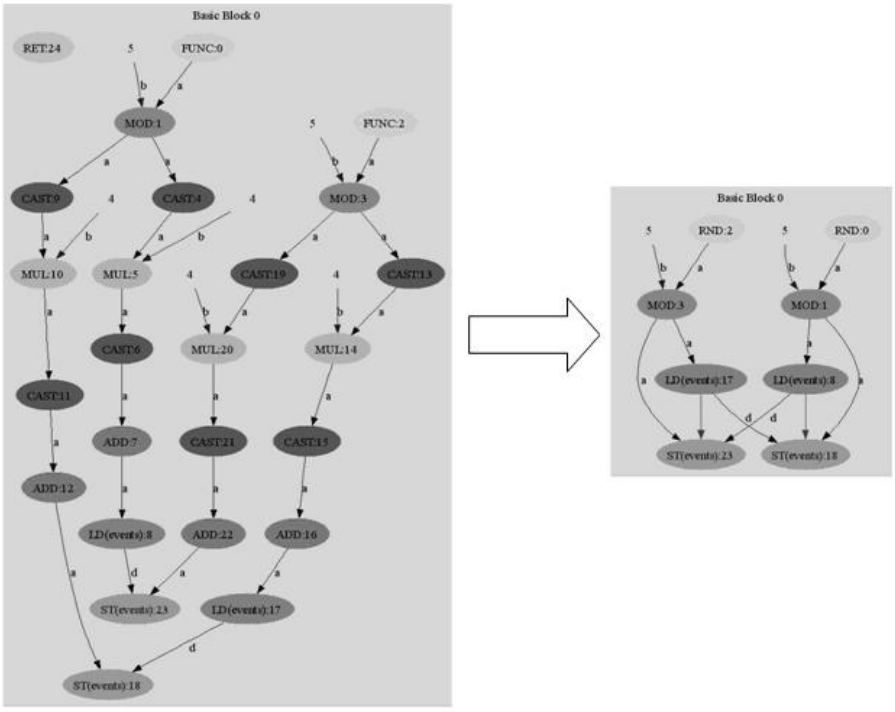


Fig. 4.5: Optimization of a CDFG.

classification from CDFG to data flow graph (DFG). At this point, the DFG is ready to be passed to the SAM algorithm.

4.5 Area and Timing Metadata

The task of a schedule and map algorithm used for architecture generation is to identify an (optimal) architecture that meets both timing and resource constraints. In this section, a method for quickly estimating FPGA resource utilization is described. Additionally, a currency is introduced for dynamically comparing relative resource costs between functional units.

4.5.1 Estimating Resource Usage

Xilinx ISE is a tool for mapping high level hardware designs written using VHDL, Verilog, or schematic onto FPGAs and other hardware devices. The basic building block of an FPGA is the look-up table (LUT). In Xilinx FPGAs, two LUTs and associated logic

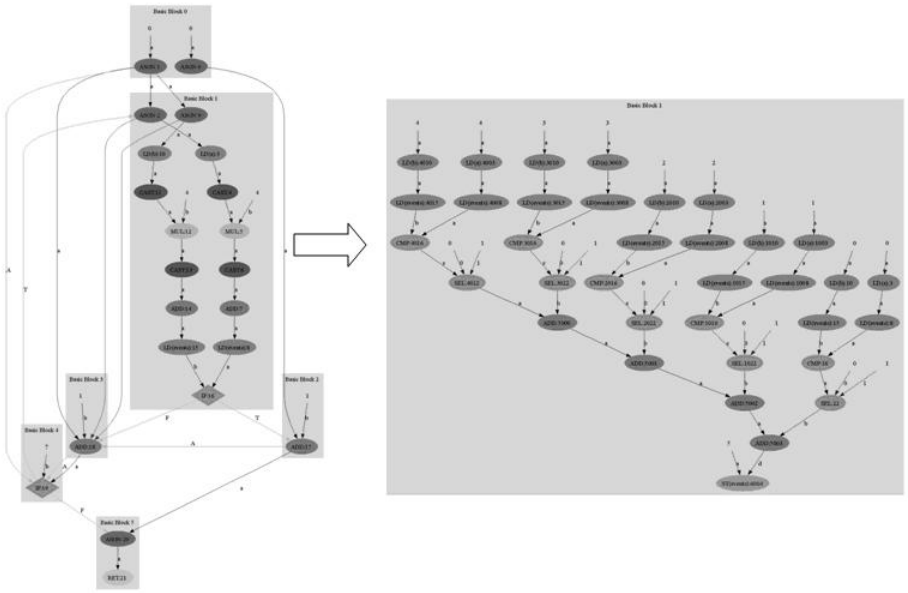


Fig. 4.6: Optimization of a more-complex CDFG. Loop unrolling and predicative execution are represented.

form what is called a slice. Xilinx FPGAs also contain embedded 18 kb block RAM units (BRAMs) and embedded multiply-accumulate ASICs (DSP48s). One of the chief concerns of a hardware designer is ensuring that the hardware design will fit on a specific chip with a finite number of resources.

Actual resource usage can be obtained from ISE by taking a design to the place-and-route stage. However, running the Xilinx tools on a large design is often time-consuming. Complex designs can take several hours to progress through the steps from synthesis to placing and routing. A mechanism for estimating resource utilization and permissible clock speeds without running these tools would greatly decrease design time.

One possible method for performing this estimation is to use curve fitting. Essentially, each type of design component is modeled for several different sizes of data inputs. Each model is sent through the Xilinx tool chain and final utilization values are determined. In this manner, a set of points are generated, where each point consists of an independent data parameter and dependent utilization parameter. A handful of these points scattered across the architecture space can be used as an input to a curve-matching algorithm (available in

Table 4.1: Measured Resource Utilization and Clock Speed for Discrete Sizes of Integer Adders

Data Width	LUTs	FFs	Maximum Clock Freq. (MHz)
1	1	1	1,381
2	2	2	1,381
4	4	4	712
8	8	8	676
12	12	12	619
16	16	16	571
24	24	24	494
32	32	32	436

Matlab), in which a high-order polynomial equation can be derived which approximates the curve represented by the design points.

For example, consider an integer addition unit. For simplicity, it is assumed that the adder takes in two n-bit numbers and produces a single n-bit output, registering the output and consuming a single clock cycle. The goal is to derive functions that relate data width (n) to resource consumption and maximum clock speed. To determine these functions, several different sizes of adders are instantiated using Xilinx ISE, noting the post-place-and-route utilization statistics for each one, including resources used and maximum allowable clock frequency. This data is shown in Table 4.1. Using this data, best-fit equations can be derived for LUT and flip-flop utilization and maximum allowable clock frequency. A fifth-order polynomial, generated by Matlab, is deemed sufficient for curve approximation. This fifth-order polynomial is shown in Eq. (4.1).

$$y = C_5 n^5 + C_4 n^4 + C_3 n^3 + C_2 n^2 + C_1 n + C_0 \tag{4.1}$$

The goal is to find the value of all coefficients (C_x). Using the Matlab curve-fitting function, the best-fit 5th-order polynomial for LUT usage in an integer adder is shown in Eq. (4.2). Coincidentally, the same equation also applies for flip-flop usage.

$$y = 0n^5 + 0n^4 + 0n^3 + 0n^2 + 1.0n + 0 \tag{4.2}$$

This equation can then be used to predict LUT and flip-flop utilization for an integer adder of arbitrary size. The correlation between the measured and computed values is shown in Fig. 4.7. In both cases it is a perfect match. A similar equation and plot can be generated

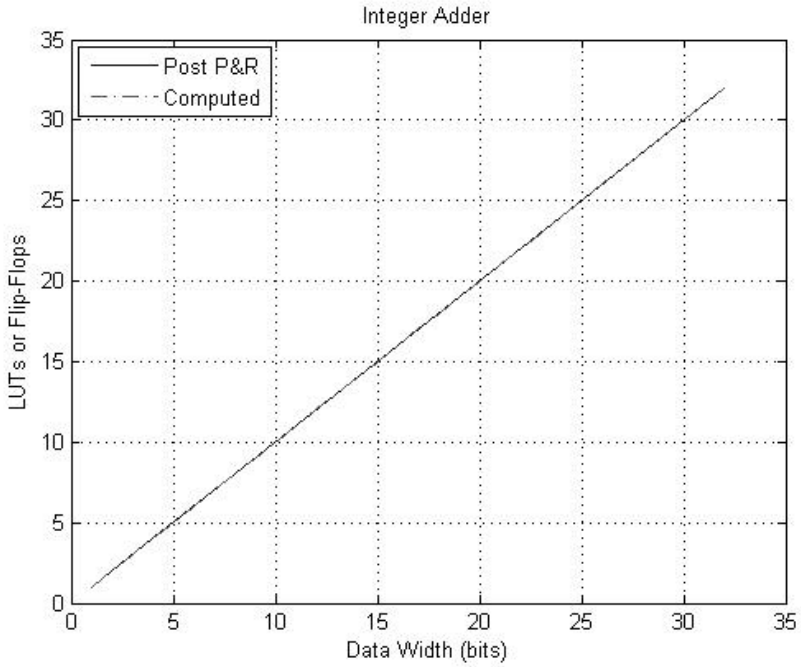


Fig. 4.7: LUT and flip-flop consumption for integer adders.

for maximum clock frequency. These are shown in Eq. (4.3) and Fig. 4.8, respectively.

$$y = -0.00122n^5 + 0.110n^4 - 3.73n^3 + 58.6n^2 + 434.0n + 1850 \tag{4.3}$$

This technique can be repeated for any module in which resource utilization is a function of data width. Table 4.2 shows the data that has been derived for integer multipliers of varying data widths, once again with a single-cycle clock latency. In addition to LUTs and flip-flops, integer multipliers can use DSP48 resources on the FPGA. Equations are derived for LUT utilization (Eq. (4.4)), flip-flop utilization (Eq. (4.5)), DSP48 utilization (Eq. (4.6)), and maximum clock frequency (Eq. (4.7)).

$$y = 0n^5 - 0.00210n^4 + 0.0669n^3 - 0.911n^2 + 4.62n - 2.92 \tag{4.4}$$

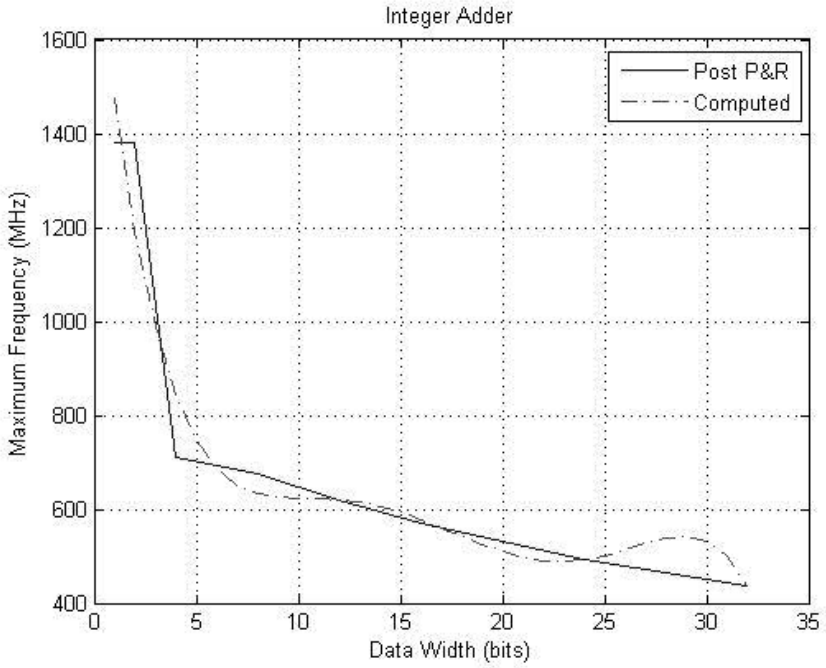


Fig. 4.8: Maximum clock frequency for integer adders.

Table 4.2: Measured Resource Utilization and Clock Speed for Discrete Sizes of Integer Multipliers

Data Width	LUTs	FFs	DSP48s	Maximum Clock Freq. (MHz)
1	1	1	0	1381
2	2	2	0	1381
4	7	4	0	461
8	0	0	1	277
12	0	0	1	277
16	0	0	1	277
24	0	24	3	128
32	0	32	3	128

$$y = 0n^5 - 0.00434n^4 + 0.135e - 1n^3 - 1.66n^2 + 7.35n - 5.86 \quad (4.5)$$

$$y = 0n^5 - 0n^4 + 0.0101n^3 - 0.124n^2 + 1.05n - 0.174 \quad (4.6)$$

$$y = 0n^5 + 0.0625n^4 - 2.90n^3 + 59.3n^2 - 536n + 1990 \quad (4.7)$$

Fig. 4.9, Fig. 4.10, Fig. 4.11, and Fig. 4.12 show the quality of the estimation for LUTs, flip-flops, DSP48 units, and maximum clock frequency, respectively. Estimations are generally very accurate. A set of equations for multiplexers of different widths can also be derived.

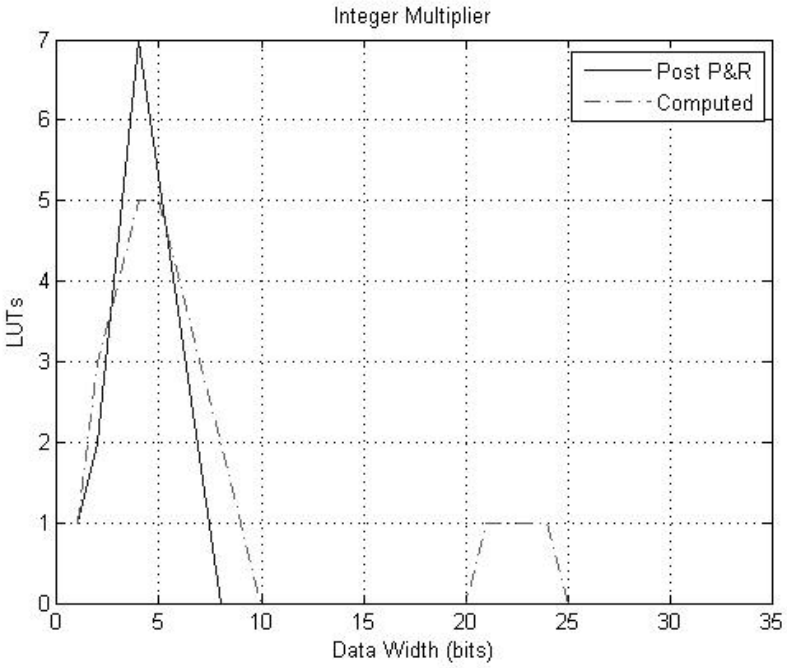


Fig. 4.9: LUT utilization for integer multiplier.

The number of inputs to a traditional multiplexer is generally a power of 2 (2, 4, 8, 16, 32, etc.). In many cases, not all input lines may be used. In theory, creating a multiplexer with 31 inputs should use the same resources as one with 32. Due to suboptimal analysis by the Xilinx tools, however, this was found to be an incorrect assumption in general. Measured LUT utilization data for 4-to-1 and 8-to-1 multiplexers is given in Table 4.3 and Table 4.4, respectively. The equations for modeling the LUT usage and maximum clock frequency of 4-to-1 multiplexers are given in Eq. (4.8) and Eq. (4.9), respectively. Similar equations for

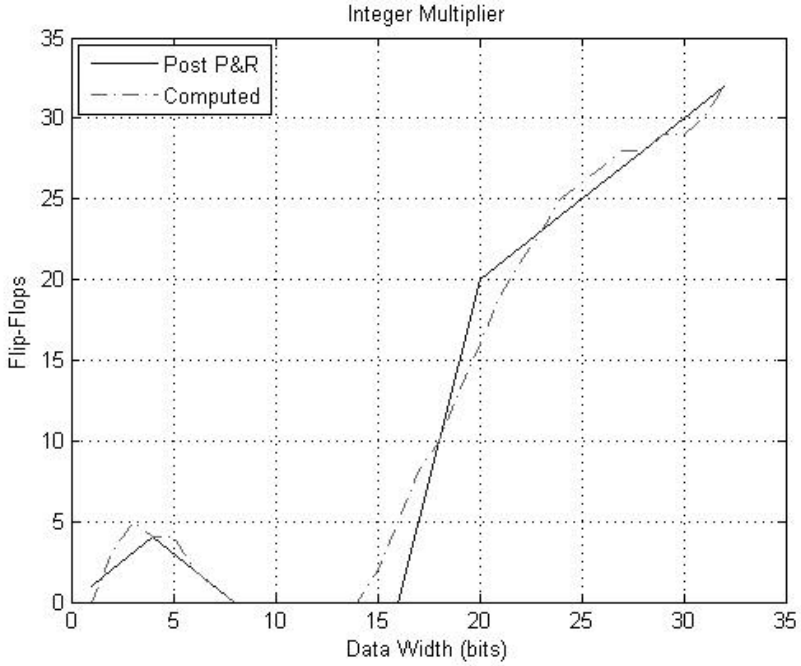


Fig. 4.10: Flip-flop utilization for integer multiplier.

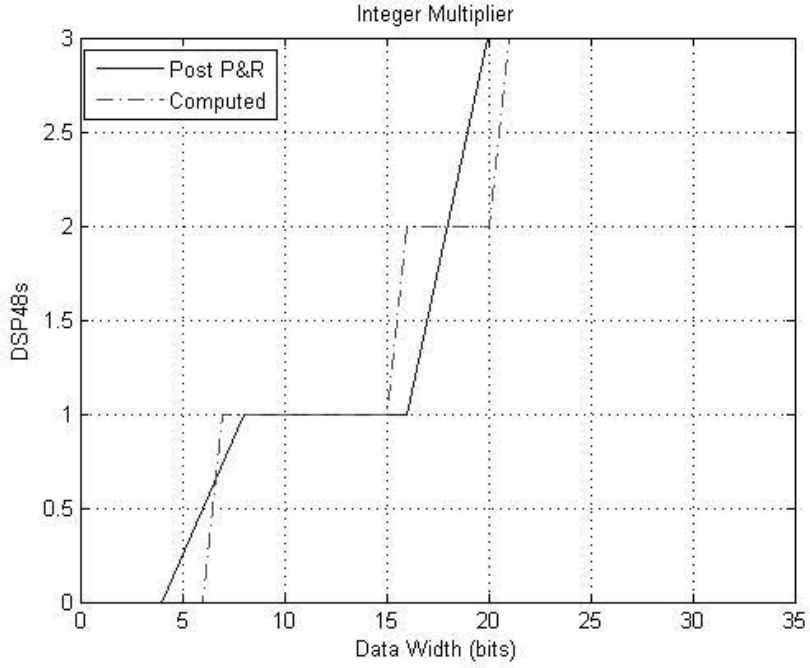


Fig. 4.11: DSP48 utilization for integer multiplier.

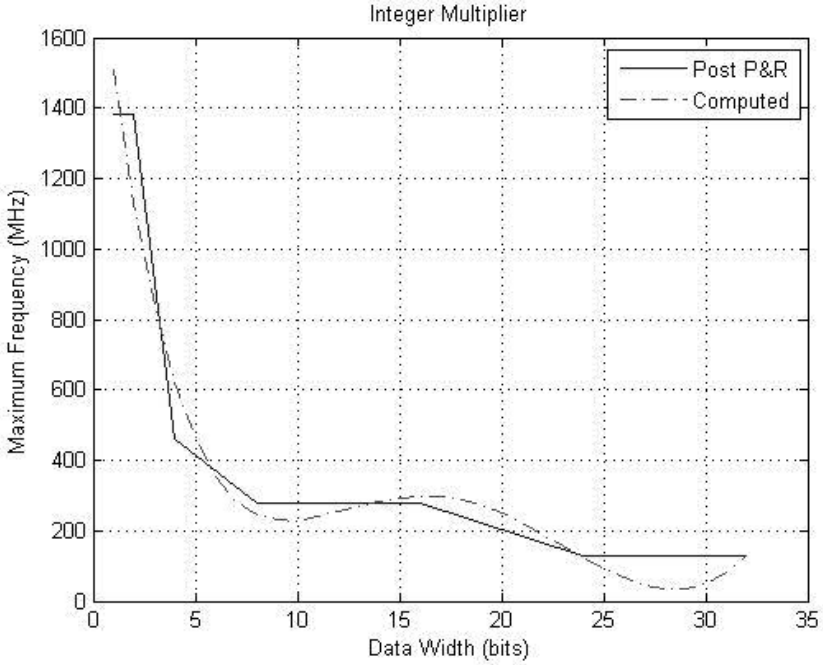


Fig. 4.12: Maximum clock frequency for integer multiplier.

Table 4.3: Measured Resource Utilization and Clock Speed for a 4-to-1 Multiplexer

Data Width	LUTs	Maximum Clock Freq. (MHz)
1	2	1377
2	4	1377
4	8	1377
8	16	1377
12	24	1377
16	32	1377
24	48	1377
32	64	1377

Table 4.4: Measured Resource Utilization and Clock Speed for an 8-to-1 Multiplexer

Data Width	LUTs	Maximum Clock Freq. (MHz)
1	4	975
2	8	975
4	16	975
8	32	975
12	48	975
16	64	975
24	96	975
32	128	975

8-to-1 multiplexers are given in Eq. (4.10) and Eq. (4.11).

$$y = 0n5 + 0n4 + 0n3 + 0n2 + 2.0n + 0 \tag{4.8}$$

$$y = 0n5 + 0n4 + 0n3 + 0n2 + 0n + 1380 \tag{4.9}$$

$$y = 0n5 + 0n4 + 0n3 + 0n2 + 4.0n + 0 \tag{4.10}$$

$$y = 0n5 + 0n4 + 0n3 + 0n2 + 0n + 975 \tag{4.11}$$

The performance of the estimation for the LUT utilization for 4-to-1 multiplexers is shown in Fig. 4.13. Fig. 4.14 shows similar data for 8-to-1 multiplexers. Estimated and measured values in all cases are an exact match. After characterization of several iterative repair

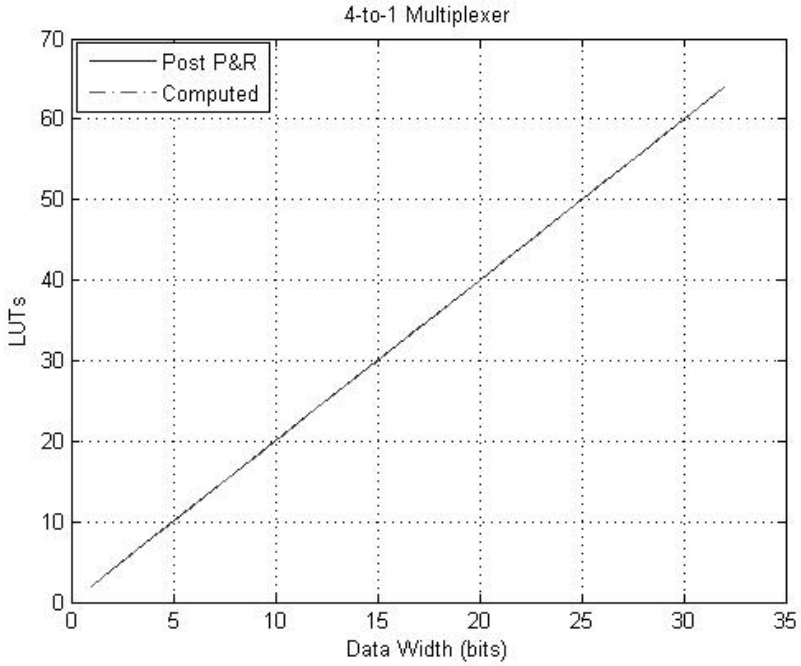


Fig. 4.13: LUT utilization for 4-to-1 multiplexers.

and other simulated annealing algorithms, a set of basic modules has been identified for parameterization. These modules are 2-to1, 4-to-1, and 8-to-1 multiplexers, registers, random number generators (RNG), constant generators, control units, adders, subtractors, multipli-

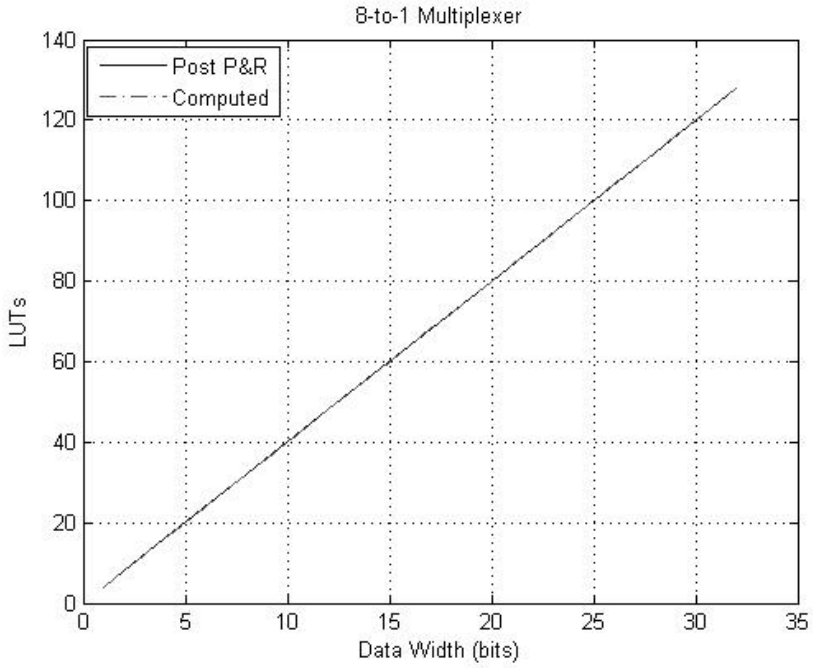


Fig. 4.14: LUT utilization for 8-to-1 multiplexers.

ers, dividers, modulus operators, absolute value (ABS) operators, and greater than, greater than or equal, less than, less than or equal, equal and not equal comparators. All modules are assigned a latency of one clock cycle, with the exception of the divider and modulus arithmetic units, which take 19 cycles, and the multiplexers, which are not clocked.

Now that the basic building blocks have been parameterized, the next question is to determine how well this method will estimate circuits comprised of two or more of these blocks in combination. In theory, resource consumption should be a roughly additive property, while maximum clock frequency should be close to that of the worst-performing block. As a simple example the circuit shown in Fig. 4.15 is considered. Using the blocks described

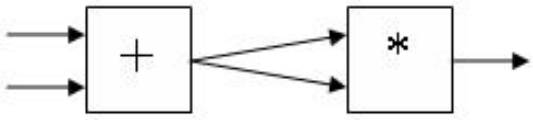


Fig. 4.15: Simple circuit with an addition operation followed by a squaring operation.

above, the circuit of Fig. 4.15 takes two clock cycles to produce a result. This circuit can

Table 4.5: Computed vs. Actual Values for Simple Add/Multiply Circuit

Data Width	LUT Usage			Flip-Flop Usage			DSP48 Usage			Max. clock Freq. (MHz)		
	Est	Act	% Err	Est	Act	% Err	Est	Act	% Err	Est	Act	% Err
4	11	13	15.4	8	8	0	0	0	0	461	354.86	29.9
8	8	8	0	8	8	0	1	1	0	277	254.13	9.0
12	12	12	0	12	12	0	1	1	0	277	254.13	9.0
16	16	16	0	16	16	0	1	1	0	277	254.13	9.0
24	24	24	0	48	48	0	3	3	0	128	111.16	15.1

be implemented with various data bit widths. Table 4.5 shows the results for bit widths ranging from 4 to 24. Notice that resource usage predictions are identical to the actual post-place-and-route values for all but one entry. The predicted maximum clock frequencies, on the other hand, are substantially higher than the actual values, especially in circuits with smaller data widths. The reason for this is that maximum clock frequency is determined by two parameters: (1) the number of LUTs on the critical path and (2) the total physical distance between LUTs on this path. For a given VHDL implementation of a module, the number of LUTs and other resources remains constant. However, the transmission delay between resources is a function of the place-and-route tool. This delay can vary drastically, depending chiefly on available chip resources. As a chip fills up, individual modules can no longer be kept physically close. Needed resources may be scattered widely across the chip. As the point of this resource estimation is to avoid the need of performing synthesis or place-and-route, estimating clock frequency is not feasible. Rather, the responsibility lies with the module designer to create a library of modules with roughly similar performance.

As another example, consider the circuit depicted in Fig. 4.16. Here, multiplexers are added to the design in Fig. 4.15. The resource utilization statistics are shown in Table 4.6. The actual resource utilization in this example is close to the estimated values, especially as the data width is increased.

Finally, a real-world circuit taken from the Alter stage of a typical simulated annealing architecture is considered. This circuit, shown in Fig. 4.17, consists of a random number generator, a constant generator, a modulus operator, a control unit, a 2-to-1 multiplexer,

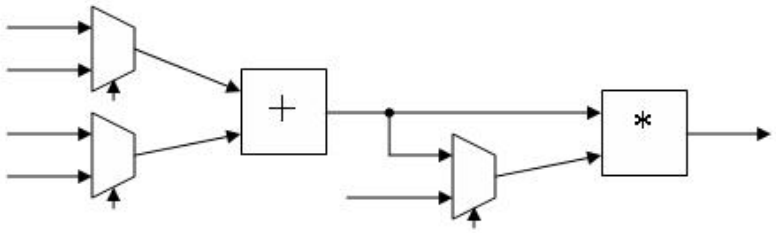


Fig. 4.16: Multiplexed add/multiply circuit.

Table 4.6: Resource Utilization for Multiplexed Add/Multiply Circuit

Data Width	LUT Usage			Flip-Flop Usage			DSP48 Usage		
	Est	Act	% Err	Est	Act	% Err	Est	Act	% Err
4	21	25	16.0	8	8	0	0	0	0
8	34	32	6.3	8	8	0	1	1	0
12	48	48	0	12	12	0	1	1	0
16	64	64	0	16	16	0	1	1	0
24	97	96	1.0	49	49	0	3	3	0

and four registers. The memory bank shown in the figure is technically external to the circuit and is not included in the resource estimation. Data width for this example is 12 bits. Table 4.7 details estimated versus actual resource usage for the circuit shown in 4.17. As maximum error is 2.7%, the estimation method is deemed valid. In conclusion, this method of approximating FPGA performance performs well for estimating resource utilization. It can be concluded that resource usage is generally an additive property for more-complex circuits built from simple parameterized blocks. Timing performance, on the

Table 4.7: Resource Usage for the Circuit Shown in 4.17

Module	Number	Estimated Resource Usage		
		LUT	Flip-Flop	DSP48
Random Number Generator	1	1	15	0
Constant Generator	1	9	12	0
2-to-1 Mux	1	12	0	0
Modulus Operator (Divider)	1	303	555	0
Control Unit	1	20	36	0
Register	4	0	48	0
Total Estimated Usage		342	666	0
Actual Usage		333	668	0
Error		2.7%	0.3%	0%

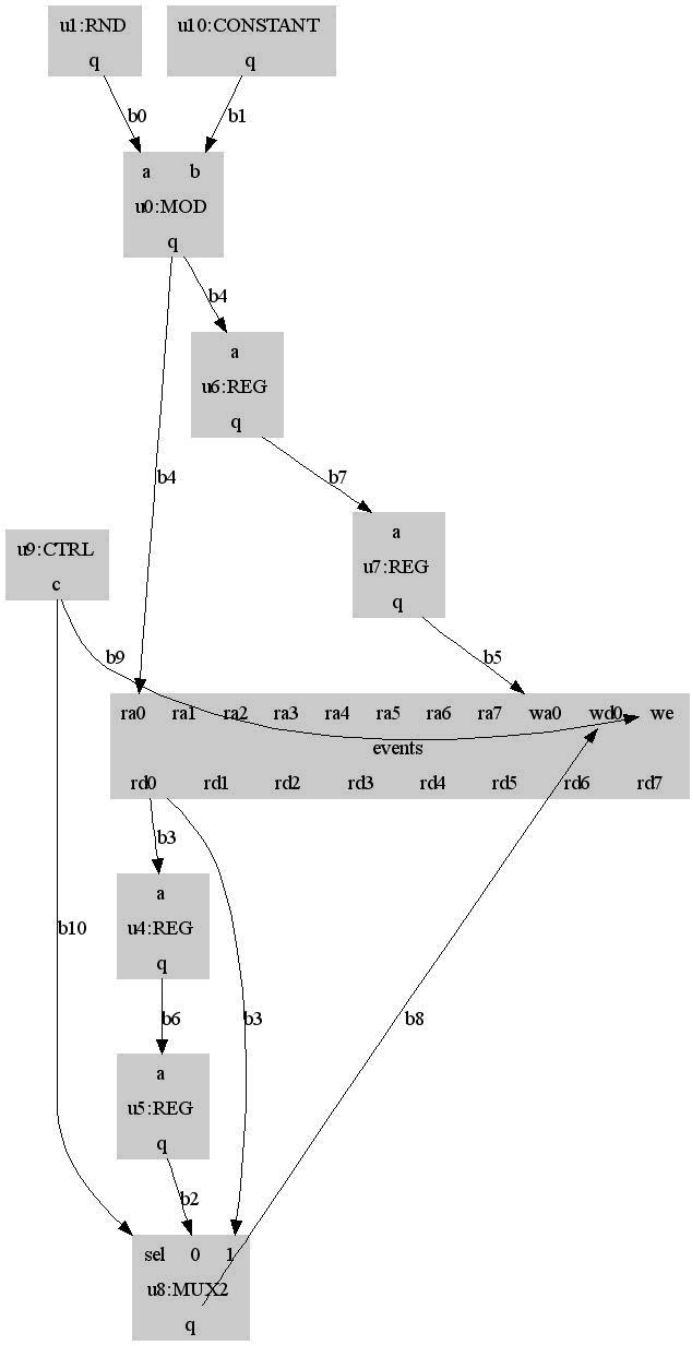


Fig. 4.17: An example architecture from a simulated annealing Alter stage.

Table 4.8: Resource Usage Data for Four Versions of a Floating-Point Multiplier

Version	LUT Usage	Flip-Flop Usage	DSP48 Usage
fmul1	641	698	0
fmul2	545	519	1
fmul3	135	224	4
fmul4	115	189	5

other hand, is not so easy to characterize. In any pipelined or clocked circuit, the maximum clock frequency is constrained by the worst-case critical path between any two registers in the circuit. In theory, the lowest maximum allowable frequency of all of the simple blocks in a circuit would be the maximum clock frequency of that circuit. However, because of competition for available resources in complex circuits, these circuits consistently perform worse than the individual components.

4.5.2 Determining Relative Resource Costs

Now that an efficient method has been described for estimating module costs, the question arises as to how to assign an area cost to a module. For example, Table 4.8 contains resource usage data for four real-world implementations of a single-precision floating-point multiplier. All four versions have a latency of eight clock cycles. Looking at these four implementations, it is impossible to determine which one is the most costly. Version one uses the most LUTs and flip-flops, but it uses the fewest DSP48s. A common currency is needed which takes into account the relative cost of LUTs, flip-flops, DSP48 ASICs, and BRAMs in a specific situation. This unit of currency is henceforth designated the RALF (RAMs, ASICs, LUTs, flip-flops).

The value of a RALF changes dynamically throughout the module instantiation phase of circuit design. RALFs are computed by summing the ratios of needed resources to available resources across all resource types. In other words, the cost of using a particular type of resource is a function of both the number of that kind of resource needed and the number available. The formal definition of a RALF is shown in Eq. (4.12) and Eq. (4.13), where U is resource usage in RALFs, r is BRAMs, a is DSP48s, l is LUTs, f is flip-flops,

Table 4.9: Comparisons Using RALFs for the Multipliers from Table 4.8

Version	LUT Usage	Flip-Flop Usage	DSP48 Usage	RALFs when available LUTs/FFs/DSP48s are		
				1000/1000/20	1000/600/50	1000/1000/10
fmul1	641	698	0	1.339	INF	1.339
fmul2	545	519	1	1.114	1.430	1.164
fmul3	135	224	4	0.559	0.663	**0.759**
fmul4	115	189	5	**0.554**	**0.593**	0.804

Table 4.10: RALF Comparisons for an Adder and a Multiplier

Version	LUT Usage	Flip-Flop Usage	DSP48 Usage	RALFs when available LUTs/FFs/DSP48s are		
				1000/1000/20	1000/600/50	1000/1000/10
add	24	24	0	**0.048**	0.072	**0.048**
mul	1	25	3	0.176	**0.057**	0.326

n_i is number of resources of type i needed, and p_i is number present. Note that the cost is infinite if adequate resources of any type are not available.

$$U_i = \frac{n_i}{p_i} \quad \text{if} \quad n_i \leq p_i \quad \text{else} \quad U_i = \infty; \quad i\epsilon\{r, a, l, f\} \tag{4.12}$$

$$U = \sum_{i\epsilon\{r,a,l,f\}} U_i \tag{4.13}$$

In Table 4.9, the four multipliers from Table 4.8 are compared using RALFs, computed for three different cases of available resources. The lowest-cost module in each situation is shown in bold. Notice that multiplier 1 cannot be implemented under one set of constraints. RALFs can also be used to compare different types of modules. For example, say a decision needs to be made between implementing an integer adder and an integer multiplier, either of which is sufficient to allow the circuit to meet timing constraints. The correct module to choose is once again a function of the number of resources consumed versus the number available for each module. Table 4.10 shows RALF values, assuming 24-bit data widths, for three different cases of available resources. Once again, looking at LUT, flip-flop, and DSP48 usage does not give sufficient information to determine which module is cheaper. As in the previous example, the cheapest module changes with the available resources. In addition to individual components, RALFs can also be used to determine the relative costs

of entire circuits. The RALF value of a circuit is simply the sum of the RALFs of each component in that circuit.

In summary, methods have been derived for both estimating the number of resources a circuit will consume and comparing circuits using the RALF. Both of these concepts play vital roles in the architecture derivation methods described in the next section.

4.6 Scheduling and Mapping (SAM)

4.6.1 Introduction

The scheduling and mapping stage (SAM) is the crux of the SATH algorithm. The task of SAM is to assign a start time (scheduling) and a resource (mapping) to each node in the optimized DFG. The goal is to find the smallest circuit that meets or exceeds a specified latency. It is assumed that suitable hardware modules exist onto which DFG nodes can be mapped. Addition nodes should be mapped to adders, multiplication nodes to multipliers, load and store nodes to memory elements, etc. The architecture space to be searched by the SAM algorithm is a combinatorial space in three dimensions: time, operation type, and resource number, as shown in Eq. (4.14), where z represents problem size, P is the permutation function, t is the time constraint, and r_i and n_i are the number of resources and the number of nodes of type i, respectively.

$$z = \prod_{i=1}^{p} P((tr_i), n_i) \tag{4.14}$$

Fig. 4.18 shows a simple example. The four-node graph shown in Fig. 4.18a consists entirely of addition nodes, thus the magnitude of the operation is one. Assume that four adders are available, an addition operation takes one time step to complete, and all operations should complete within four time steps. Fig. 4.18b and Fig. 4.18c show two legal placements for all nodes in the two-dimensional matrix. These placements are considered legal as no node dependences are violated, meaning no child node is scheduled to begin before all associated parent nodes have completed. Fig. 4.18b represents a configuration that uses four adders

and completes in three time steps; while Fig. 4.18c uses a single adder, completing in four time steps. A score can be assigned to the different architectures, depending on the relative importance of resource usage and execution time. The size of the search space shown in

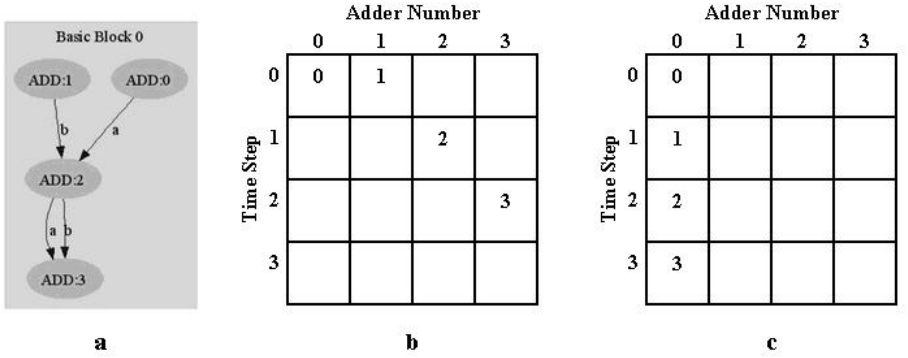


Fig. 4.18: Possible schedules and mappings for a simple DFG. The DFG shown in (a) can be scheduled and mapped as shown in (b) or (c).

Fig. 4.18 may appear small, but is in truth quite large. Strictly speaking, the search space is the number of permutations of four nodes scattered across a four-by-four grid. This yields 43,680 possible configurations. However, techniques can be used to trim the size of the search space. First, the number of available hardware units need not be larger than the number of nodes to be mapped. In the example of Fig. 4.18 there are four addition operations. Thus, the largest number of adders needed is four, as any additional adders would simply go unused. Second, the number of available time steps can be bounded by the sum of the latencies of all operations in the DFG. The total latency of all additions in Fig. 4.18 (assuming one time step per addition) is four. Any additional time steps would be wasted time with nothing scheduled. Third, the window of time steps in which each node can be scheduled can be constrained by computing the ASAP (As Soon As Possible) and ALAP (As Late As Possible) values for each node. This will remove a large number of illegal solution states from the search space. ASAP and ALAP schedules are simple to compute. ASAP scheduling starts with source nodes, which are assigned a start time of zero. The tree is then traversed from top to bottom, with each node being assigned the earliest possible start time that does not violate dependences from parent nodes. ALAP

Table 4.11: ASAP and ALAP Schedules for the SAM Problem Shown in Fig. 4.18

Node	ASAP Time Step	ALAP Time Step
0	0	1
1	0	1
2	1	2
3	2	3

scheduling is similar. All sink nodes are assigned the latest possible start time, based upon the target latency of the entire graph and the individual computation latency of each sink node. The graph is then traversed from bottom to top, with each node being assigned the latest possible start time that does not violate dependences to child nodes. Table 4.11 lists ASAP and ALAP times for the SAM problem shown in Fig. 4.18. Given these added constraints, the size of the search space can be formally described by the equation shown in Eq. (4.15), where s is the number of states in the search space, n is the number of nodes in the DFG, r_i is the number of nodes of type i, and t_{ALAP} and t_{ASAP} are the ALAP and ASAP time boundaries, respectively.

$$s = \prod_{i=1}^{n} r_i(t_{ALAP_i} - t_{ASAP_i}) \tag{4.15}$$

Using this equation and the information from Table 4.11, the size of the search space for the problem shown in Fig. 4.18 is reduced from 43,680 states to 4,096 states, a reduction of over ten times. The size of the search space becomes critical for large DFGs with hundreds or thousands of nodes. Search space sizes for these problems can easily exceed billions or even trillions of states. Appropriately trimming the architecture space can potentially reduce search times from days to minutes.

A typical evaluate stage DFG for the SAM stage to process is shown in Fig. 4.19. This DFG consists of 1,783 total nodes. The type and number of nodes in Fig. 4.19 are listed in Table 4.12. Assuming that each node in the DFG consumes one time step, the graph in fig. 50 has a minimum total latency of 106 time steps. If the target latency for a valid architecture is 120 time steps, the number of states in the architecture space would be $7.25(10^{24})$ states using the nave permutation method. Using the constraints specified

Table 4.12: Node Distribution by Type for the DFG Shown in Fig. 4.19

Node Type	Number Present in DFG
add	197
subtract	198
absolute value	198
load	792
store	1
constant	397

by Eq. (4.15), the size of the space is reduced to $6.79(10^{20})$ states, a reduction in size of $1.07(10^{4})$ times.

Fig. 4.19: A typical evaluate stage DFG for evaluation by the SAM algorithm.

4.6.2 Possible SAM Algorithms

Based upon the above discussion of the combinatorial nature of the size of the search space for the schedule and map problem, an efficient SAM algorithm is imperative to provide results within a reasonable amount of time. Chapter 2 described several techniques for

exploring the architecture space. Three different exploration methods were implemented and evaluated, each varying in execution time and quality of results.

The first method attempted is simple simulated annealing using iterative repair. The algorithm begins by allocating a resource number and a time step to each node in the DFG. Architectural exploration is done by picking a random node from the list, modifying either the time step or the assigned resource number of that node, and computing a score for the new configuration. Components of the score include penalties for the number of dependency violations and a lesser penalty for the overall size of the circuit, computed using the resource estimation technique described in this chapter. The computation of the circuit size is complicated by the insertion of multiplexers and delay registers, which also consume resources and must be factored in to the overall resource usage cost. If multiple DFG nodes are mapped to the same resource at different time steps, a multiplexer is generally needed to select the correct input at the correct clock cycle. If there is a delay of one or more cycles between the completion of execution of a parent node and the commencement of execution of a child node, one or more delay registers must be inserted to preserve correct circuit timing. As an example, consider the DFG and associated architecture shown in Fig. 4.20. Notice that the DFG shown in Fig. 4.20a consists of two constants, two random number generations (RND), two modulus nodes (MOD), two loads (LD), and two stores (ST). The architecture, shown in Fig. 4.20b, only has one random number generator (RND), one constant generator (CONSTANT), one divider (MOD), and one write port. Three registers (REG) and one two-to-one multiplexer (MUX2) are needed to manage dataflow through the circuit. Notice that a controller module (CTRL) is also present in the architecture to control the multiplexer select line and the memory write-enable line. The resource cost of registers and multiplexers cannot be neglected in overall resource usage computations, as multiplexers can be more expensive than some operational units.

Unfortunately, the simple simulated annealing method for SAM proves to be intractable for problems of any significant size. The example discussed in Fig. 4.19 and Table 4.12 consists of 6.79x1020 states, following Eq. (4.15). While the strength of simulated annealing

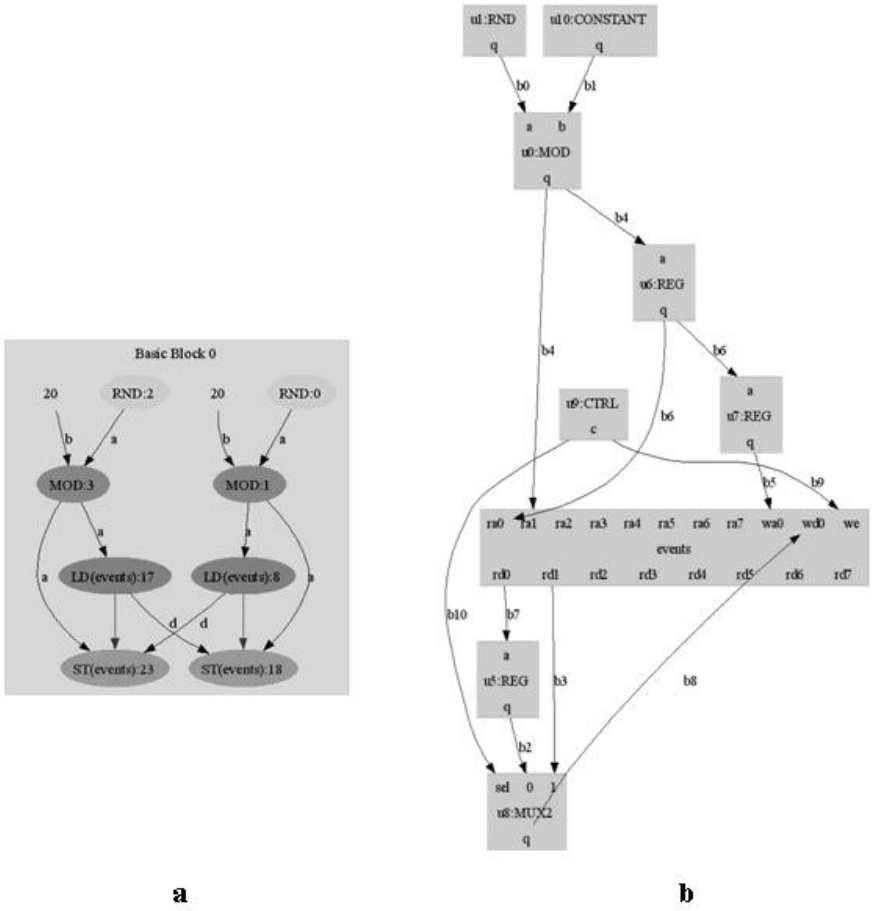


Fig. 4.20: Example of a DFG (a) with an associated architecture (b), including a multiplexer and three delay registers.

lies in the fact that a close-to-optimal solution can be found while only visiting a fraction of the states in the architecture space, it is still limited in applicability to very large spaces. For example, say that only one percent of the total states in the space need to be visited to find a good solution and that the SAM stage can evaluate 10,000 solution states per second. The problem described by Fig. 4.19 and Table 4.12 would require $6.79(10^{14})$ seconds, or about 21.5 million years to evaluate. Thus, it can be concluded that an all-in-one heuristic search is not a viable solution.

The second method attempted is Force Directed Scheduling (FDS) [47, 48]. FDS is described in chapter 2. As a quick summary, FDS is not a search algorithm, but rather a mathematical approximation algorithm. The FDS algorithm loops only once for each node in the DFG, meaning even the most-complex graphs can be scheduled and mapped in a matter of seconds. On each iteration, FDS computes a force for each unscheduled node and then fixes the node/cycle pair with the lowest force. Components of the force equation include self force, predecessor force, and successor force. The strategy behind FDS is to reduce the concurrency of nodes of the same type while meeting the specified circuit latency. It is important to note that FDS is a scheduling algorithm only, and must be coupled with a mapping algorithm to complete the schedule-and-map task. Unfortunately, the one-and-done nature of FDS, coupled with the lack of provision for estimating needed multiplexer and register resources, leads to suboptimal results for DFGs of any significant size. Extremely wide multiplexers with 32 or 64 inputs are often required to manage data flow. These wide multiplexers are inefficient in resource usage and slow down maximum clock speeds. For example, a 64-to-1 multiplexer handling 32-bit data lines consumes 1024 LUTs on a Virtex 4 FPGA. Additionally, the greedy nature of FDS often results in illegal architectures. For example, scheduling can take place in such a way that several write operations must be performed on the same memory on the same clock cycle, requiring several write ports. The physical properties of Virtex 4 FPGAs limit the number of write ports on any memory block to two.

4.6.3 SAM Method of Choice

As discussed previously, appropriately trimming the architecture space can potentially reduce search times from days to minutes. Even with smart management of the architecture space, the size of the space easily exceeds manageable levels. A traditional way to reduce the size of the space is to break the problem into two parts: scheduling and mapping. Scheduling involves assigning each node in a DFG to a specific start time. Mapping means assigning each scheduled node to a specific resource. SATH makes use of this partitioning of work. Two simulated annealing algorithms are executed, one to schedule the DFG and a second to map. A simulated annealing algorithm is adapted to solving a specific type of problem through customization of five parameters: solution format, starting solution, alteration strategy, evaluation strategy, and temperature manipulation. These characteristics are defined for both the scheduling and mapping algorithms below.

The task of the scheduler is to assign a start time to each node in the DFG. A logical representation of a solution is an array, where index represents node ID and value represents start time. Determining a correct initial solution is critical to efficient execution of the algorithm. The initial solution for the SATH scheduler is the ASAP start time for each node, thus ensuring that the algorithm begins with a valid schedule. The solution alteration strategy is to randomly pick one location in the array and reassign the associated start time to a random number that lies within the window bounded by the ASAP and ALAP times.

According to the pseudocode in Fig. 2.8, once a new solution has been produced by altering the prior solution, it must be evaluated. This equation is given in Eq. (4.16) through Eq. (4.21).

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{p-1} s_j + t_j - s_i \quad \text{if} \quad s_j + t_j > s_i \quad \text{else} \quad 0 \tag{4.16}$$

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{p-1} s_i - (s_j + t_j) - 1 \quad \text{if} \quad s_j + t_j < s_i \quad \text{else} \quad 0 \tag{4.17}$$

$$\forall c_{r \epsilon R, s \epsilon S} \sum_{i=0}^{n-1} 1 \quad \text{if} \quad r_i = r \quad \text{and} \quad s_i = s \quad \text{else} \quad 0 \tag{4.18}$$

$$\forall w_{r \epsilon R} \forall s \epsilon S(w_r = max(w_r, c_{r,s})) \tag{4.19}$$

$$u = \sum_{i=0}^{r-1} w_i a_i \tag{4.20}$$

$$q = 100v + u + a_{reg} d \tag{4.21}$$

From Eq. (4.16), v is the sum of the magnitude of all cases in which a node (in set n) is scheduled such that one or more of its parent nodes (in set p) have not yet completed, where s represents start time and t represents latency. Register usage is determined using Eq. (4.17). Any time a gap of one or more cycles exists between completion of a parent node and commencement of a child node, registers are needed. Because mapping has not yet occurred, worst-case register usage is computed, assuming each node is mapped to a unique resource and no register sharing exists. In Eq. (4.18) through Eq. (4.20), the circuit size (less registers and multiplexers) is computed. In Eq. (4.18), a concurrency matrix, C, is assumed with dimensions equal to number of resource types (R) and target schedule length (S). The maximum number of operations of each type that start on each clock cycle is computed. In Eq. (4.19), the worst-case concurrency for each resource type, w_r is identified. In Eq. (4.20), resource usage is summed across all operation types. The score, given in Eq. (4.21), is a function of dependency graph violations (v), resource usage in RALFs (u), and estimated register usage (d) multiplied by the area cost of a register (a_{reg}). Because mapping has not yet taken place, the resource usage is an estimate derived from the concurrency of each node type in the DFG. The initial temperature of the scheduler is set using Eq. (4.22), as both the number of nodes and the number of available time slots define the size of the search space.

$$t_i = (nl)^{0.8} \tag{4.22}$$

Here, t_i is initial temperature, n is number of nodes in the DFG, and l is target latency. Through trial and error, it was found that raising this product to the 0.8 power works well across different problem sizes.

Once the scheduler has completed, the scheduled DFG is passed to the mapper. The

job of the mapper is to assign each node in the DFG to a specific resource. A mapped solution is represented as an array where index represents node ID and value represents resource number. The initial solution for the SATH mapper is a valid mapping. Each node is scheduled on the lowest-numbered available resource for the given clock cycle. The alteration strategy is to select one node at random and change the associated binding to a new, randomly selected resource. The process of assigning a score to a solution in the mapper is more complex than in the scheduler. The mapper must be cognizant of total circuit size. The size of a circuit consists of functional units and data control units (multiplexers and registers), which are used to control data flow throughout the circuit. Altering a mapping can affect both the number of functional units needed and the number of data control units needed. The evaluate equations are given in Eq. (4.23) and Eq. (4.24):

$$\rho = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \quad \text{if} \quad s_i = s_j \quad \text{and} \quad b_i = b_j \quad \text{else} \quad 0 \tag{4.23}$$

$$q = 10\rho + u \tag{4.24}$$

Here, q is solution quality, ρ is the number of resource over-usage occurrences (situations where two nodes are scheduled on the same unit at the same time, where b represents resource binding), and u is the number of resources needed (functional and control), as computed in Eq. (4.20). The search space for the mapping problem is a function of the size of the DFG. The initial temperature is computed using Eq. (4.25).

$$t_i = n^{1.2} \tag{4.25}$$

In general the SAM stage can only operate on pipeline stages in which parallelism can be extracted. In the case of the simulated annealing problems discussed in this dissertation, the *copy*, *alter*, and *evaluate* stages can be accelerated. The *accept* and *adjust temperature* stages cannot, as they are purely sequential processes. The *copy* stage is not compute intensive but is memory intensive. Its job is to copy a solution from one memory bank to another. Techniques for accelerating the execution of this stage were described in Chapter

3.

Fig. 4.21 shows the general flow of execution for the SAM stage as it iteratively attempts to improve pipeline performance. A pipelined processor can only run as fast as the latency of the slowest stage. The algorithm iteratively reduces the latency of the worst stage until no more parallelism can be derived or FPGA resources are exhausted.

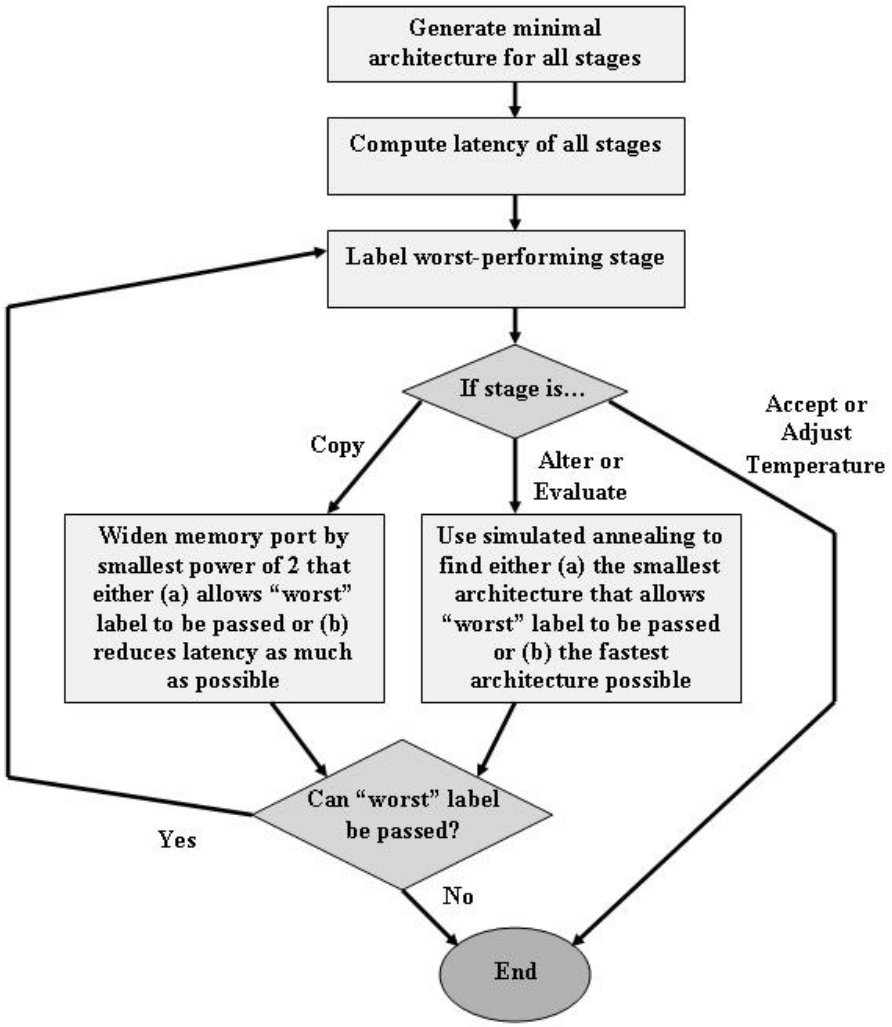
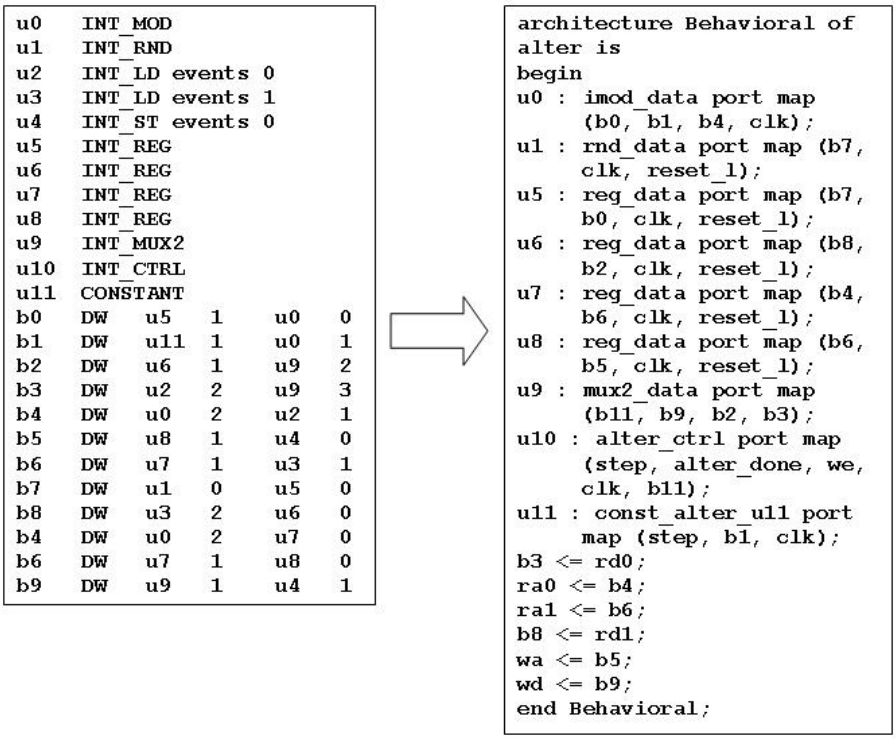


Fig. 4.21: SAM execution flowchart.

4.7 HIF to VHDL Converter

The SAM stage produces a circuit description in a custom hardware intermediate format (HIF). These files must be converted to VHDL then imported into a third-party design tool such as Xilinx ISE for synthesis and implementation. Fig. 4.22 shows an example of HIF code and equivalent VHDL. HIF format specifies modules, ports, and connections that are easily translated into VHDL.



```
u0   INT_MOD
u1   INT_RND
u2   INT_LD events 0
u3   INT_LD events 1
u4   INT_ST events 0
u5   INT_REG
u6   INT_REG
u7   INT_REG
u8   INT_REG
u9   INT_MUX2
u10  INT_CTRL
u11  CONSTANT
b0   DW   u5   1   u0   0
b1   DW   u11  1   u0   1
b2   DW   u6   1   u9   2
b3   DW   u2   2   u9   3
b4   DW   u0   2   u2   1
b5   DW   u8   1   u4   0
b6   DW   u7   1   u3   1
b7   DW   u1   0   u5   0
b8   DW   u3   2   u6   0
b4   DW   u0   2   u7   0
b6   DW   u7   1   u8   0
b9   DW   u9   1   u4   1
```

```
architecture Behavioral of
alter is
begin
u0 : imod_data port map
     (b0, b1, b4, clk);
u1 : rnd_data port map (b7,
     clk, reset_l);
u5 : reg_data port map (b7,
     b0, clk, reset_l);
u6 : reg_data port map (b8,
     b2, clk, reset_l);
u7 : reg_data port map (b4,
     b6, clk, reset_l);
u8 : reg_data port map (b6,
     b5, clk, reset_l);
u9 : mux2_data port map
     (b11, b9, b2, b3);
u10 : alter_ctrl port map
     (step, alter_done, we,
     clk, b11);
u11 : const_alter_u11 port
     map (step, b1, clk);
b3 <= rd0;
ra0 <= b4;
ra1 <= b6;
b8 <= rd1;
wa <= b5;
wd <= b9;
end Behavioral;
```

Fig. 4.22: Conversion of HIF to VHDL.

Chapter 5

Results

5.1 Test Cases

In order to test the performance of the SATH algorithm, a variety of search problems solved using simulated annealing should be tried. This section describes three such problems, all of which are directly related to space mission planning and scheduling. These problems are traveling salesperson, graph coloring, and dependency-graph violation removal.

5.1.1 Traveling Salesperson

The traveling salesperson problem (TSP) [83] consists of a set of cities that a salesperson must visit. The salesperson can move from any city to any other city. The goal of the problem is to visit every city exactly once with a minimal amount of travel. In the space environment, TSP cities could represent points of interest that an unmanned rover might want to visit while conserving the battery as much as possible.

TSP can be represented as an ordered list of cities. Fig. 5.1 shows an example for ten cities. The connections show two possible orders of visitation, either starting at node six or starting at node eight. For a problem of n cities, there are $n!$ unique visitation paths, thus making TSP an ideal problem to be solved by simulated annealing. Three features

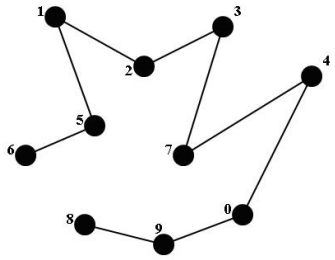


Fig. 5.1: Example traveling salesperson problem for ten cities. The edges represent two possible orders of visitation.

distinguish one simulated annealing problem from another: solution format, alter strategy, and evaluation equation. For TSP, the format of the solution is a list of cities arranged in the order to be visited. The alter strategy involves picking two locations in the solution at random and swapping the contents of those locations. Fig. 5.2 shows the solution format and alter strategy for the problem of Fig. 5.1. The initial solution indicates that cities should be visited in the order six, five, one, two, three, four, etc. Following the alter strategy, two slots in the ordering are selected at random and contents are swapped. The new ordering is six, five, four, two, three, one, etc. Once the solution as been altered, it must also be

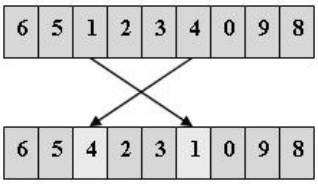


Fig. 5.2: Solution format and alter strategy for TSP.

evaluated. The score of a solution is a representation of how well it meets the goal of the problem. In the case of TSP, the goal is minimization of total distance traveled. In Fig. 5.2, this is computed by summing the distance from six to five, from five to four, from four to two, etc. Physical coordinates are provided for each city. If the goal is to minimize total Manhattan distance in two dimensions, the evaluation equation is given in Eq. (5.1), where d represents total distance, n is number of cities, and x and y represent coordinates on a two-dimensional grid. An example of TSP code can be found in Appendix B.

$$d = \sum_{i=0}^{n-2} |x_i - x_{i+1}| + |y_i - y_{i+1}| \tag{5.1}$$

5.1.2 Graph Coloring Problem

The graph coloring problem (GCP) [83] is solved by determining whether an arbitrary map in d dimensions can be shaded by region using c colors with no adjacent regions sharing the same color. This is applicable in the field of space processing as GCP can be used to determine variable or event liveness, thus indicating the number of registers or other

resources that a system needs.

The solution format for GCP is a numbered list of regions. Colors are represented as a set of unique integers. A separate list of adjacencies is also maintained. Fig. 5.3 shows how GCP can be abstracted as an adjacency graph. The problem shown in Fig. 5.3 consists of 20 regions, each represented by a node in the graph. Edges in the graph represent adjacencies. For example, region six is adjacent to regions 17 and zero. To satisfy graph colorability, neither region 17 nor zero can be assigned the same color as region six. The alter strategy

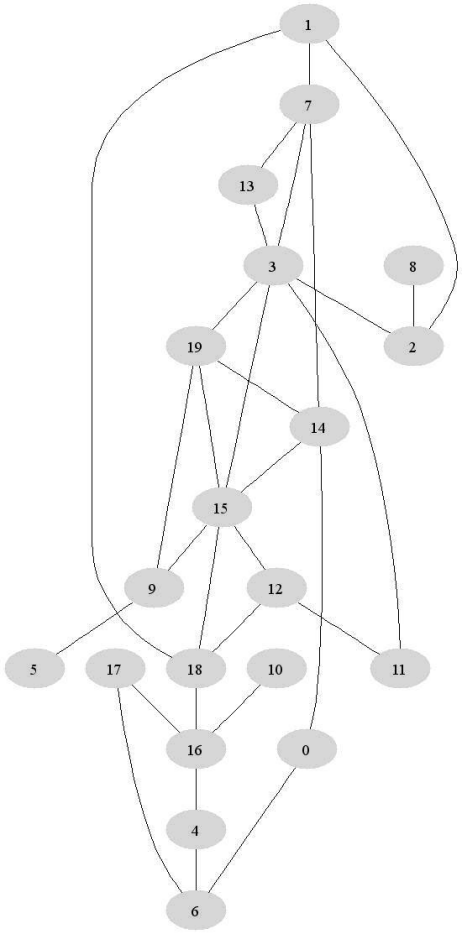


Fig. 5.3: An adjacency graph for GCP consisting of 20 regions.

for GCP is shown in Fig. 5.4. A region is selected at random and its associated color is randomly changed to a new value. Fig. 5.4 assumes nine regions and four colors, numbered zero to three. The evaluation equation for GCP is fairly simple. The goal of GCP is to

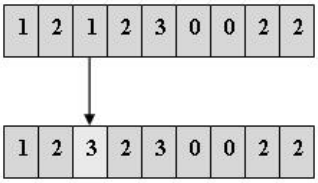


Fig. 5.4: Solution format and alter strategy for GCP.

remove all adjacency violations. The evaluate equation shown in Eq. (5.2) simply counts the number of violations in a solution, where v represents number of violations, p is the number of adjacency pairs, and a and b represent nodes connected by an adjacency edge. The assert operator returns a one if the expression is true and a zero otherwise. An example of GC code can be found in Appendix C.

$$v = \sum_{i=0}^{p-1} assert(c_{a_i} = c_{b_i}) \tag{5.2}$$

5.1.3 Dependency Graph Violation Removal Problem

The dependency graph violation removal problem (DGVRP) is a simple scheduling technique. Given a dependency graph, the job is to schedule all nodes within a constrained time period such that all parent nodes complete operation before associated children nodes begin. Space missions often include sets of dependent tasks that need to be performed with limited time in which to complete everything.

Fig. 5.5 shows an example of a dependency graph. To avoid violations, task seven must wait for tasks two and four to complete, task 12 must wait for tasks zero, three, and six to complete, etc. The DGVRP solution format is a list of start times, one for each node in the graph. The alter strategy is similar to that for GCP. A task is selected at random and its start time is changed to a new random value. The evaluate equation accumulates the magnitude of each violation. This is described in Eq. (5.3), where v is number of violations, p is the number of dependency edges, s represents a source task, d a destination task, and

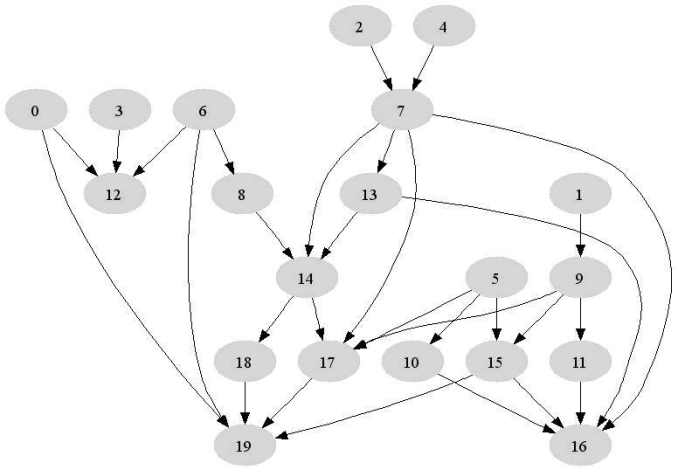


Fig. 5.5: Example of a dependency graph. Nodes represent tasks and edges represent dependencies.

t a start time. An example of DGV code can be found in Appendix D.

$$v = \sum_{i=0}^{p-1} (assert(t_{s_i} \geq t_{d_i}))(t_{s_i} - t_{d_i}) \tag{5.3}$$

5.2 SATH vs. Impulse/PPC/X86

Test cases are six different, randomly-generated examples: 20-city TSP, 100-city TSP, 20-node GCP, 100-node GCP, 20-event DGVRP, and 100-node DGVRP. All tests were performed targeting a Xilinx V4SX35 FPGA. Execution platforms considered include circuits generated by the Impulse tool, a PowerPC 750, an AMD Athlon 64 X2, and custom circuits generated by SATH. In addition to the Impulse tool for C-to-hardware conversion, attempts were made to utilize Handel-C [23], SPARK [28], and System C [18]. Josh Templin assisted in the evaluation of these tools, with the goal of identifying tools in which pure C source code is supported with little or no modification by the user. Handel-C required user-defined modules for supporting floating-point arithmetic. SPARK is designed specifically to analyze loops and branch conditions and also does not support floating-point arithmetic. System C is technically a C++ library. Source code needs to be rewritten using this library for proper functionality. Only Impulse provides a (relatively) direct path for interpreting C code with little or no modification.

Fig. 5.6 contains execution time results for all six problems when respective architectures are clocked at maximum clock rates (Impulse 121 MHz, PowerPC 373.5 MHz, AMD 2.61 GHz, SATH 175 to 183 MHz). Note the y-axis in Fig. 5.6 is a logarithmic scale. SATH significantly outperforms all platforms except for the AMD. Fig. 5.7 shows speedup of SATH-generated circuits with respect to other circuits for all cases. Speedups range from just over one to around 42 times. In many embedded applications, such as space systems,

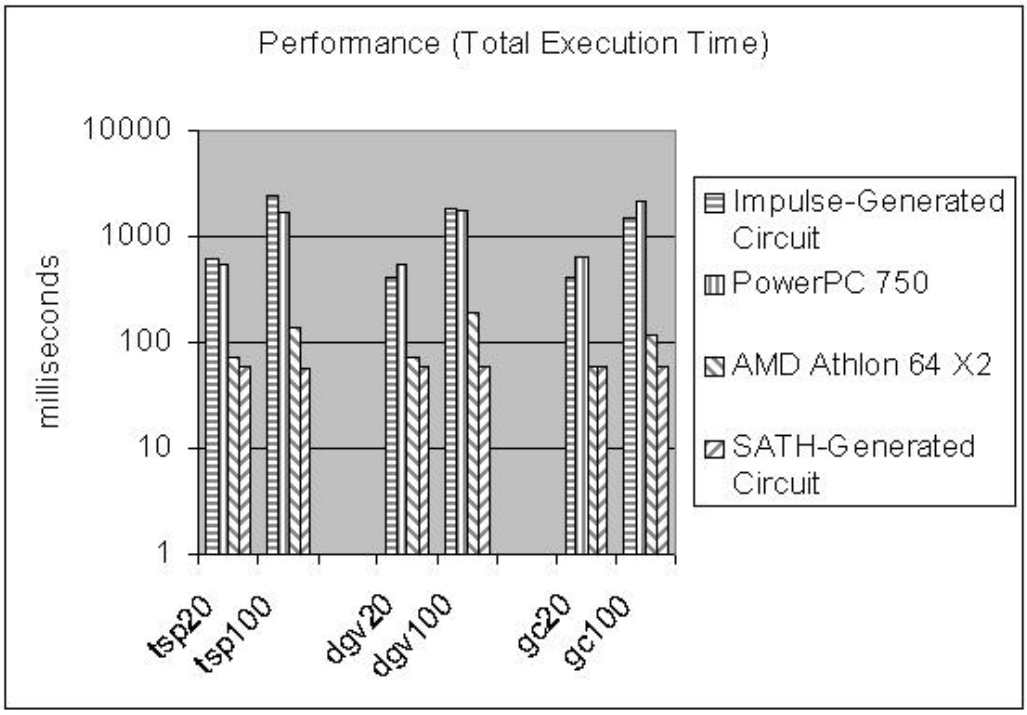


Fig. 5.6: Comparative performance in total execution time.

power usage is the principle concern in determining how fast to clock a circuit. Thus, normalizing clock frequency and comparing execution speeds in number of clock cycles may be a better measure of relative performance. Fig. 5.8 shows execution times in clock cycles per iteration and Fig. 5.9 shows associated SATH speedup. In this case, SATH-generated architectures run between five and 80 times faster than other implementations, which is a significant speedup. Finally, the performance of SATH-generated circuits in resource usage can be compared with that of other FPGA-based solutions, namely Impulse-generated circuits. Fig. 5.10 to Fig. 5.14 show this comparison with resource usage represented in LUTs

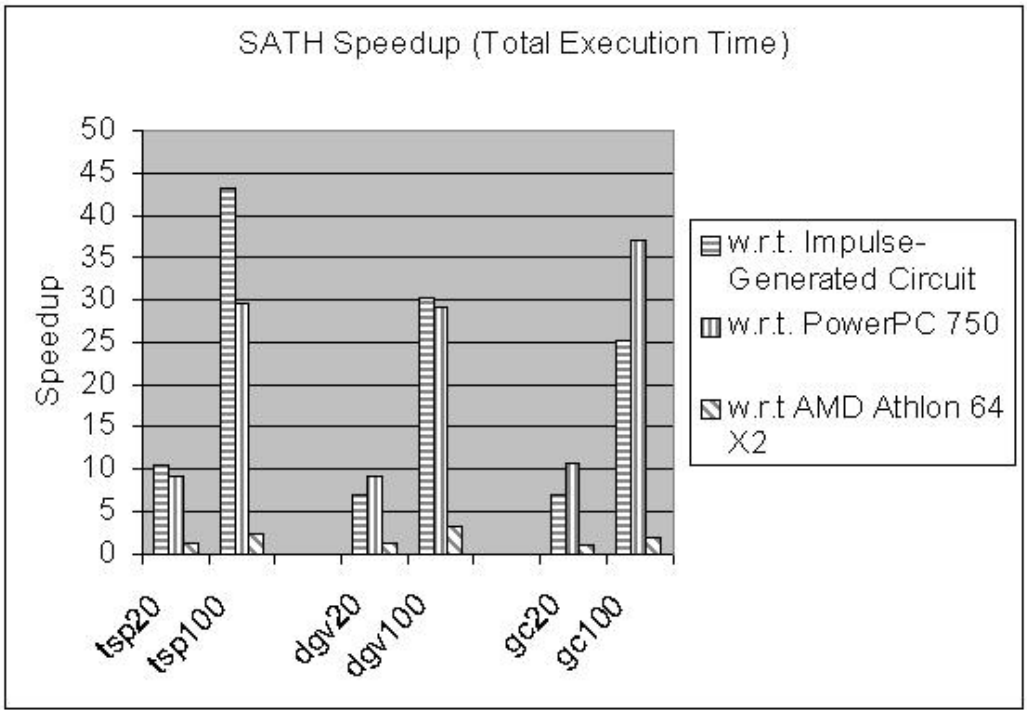


Fig. 5.7: Speedup of SATH-generated circuits comparing total execution time with respect to other implementations.

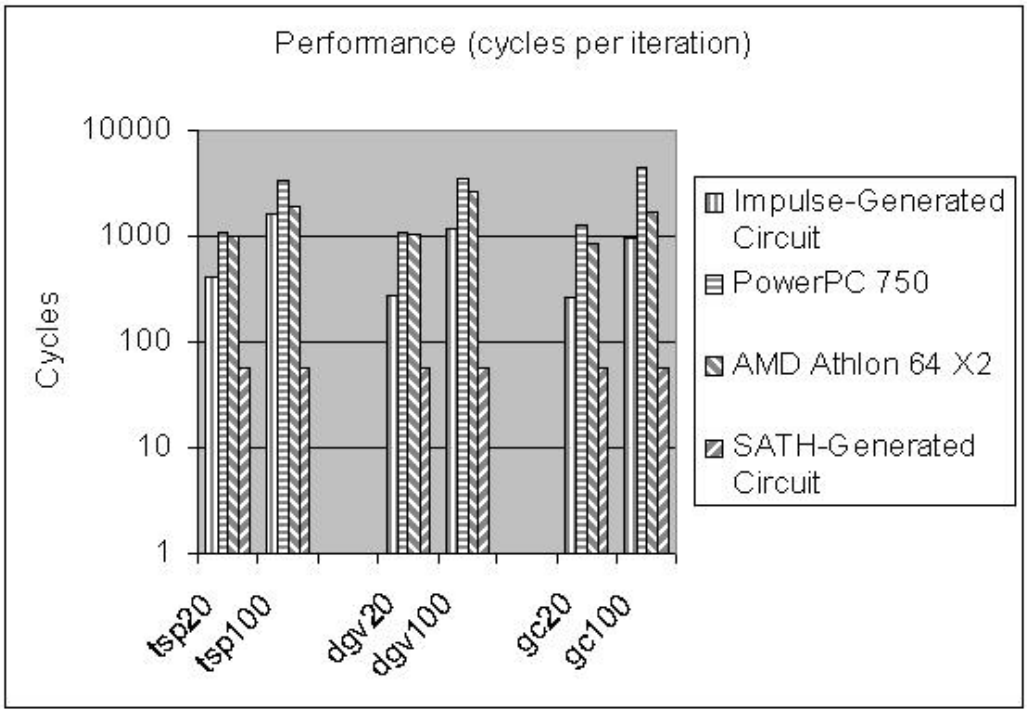


Fig. 5.8: Comparative performance in cycles per iteration.

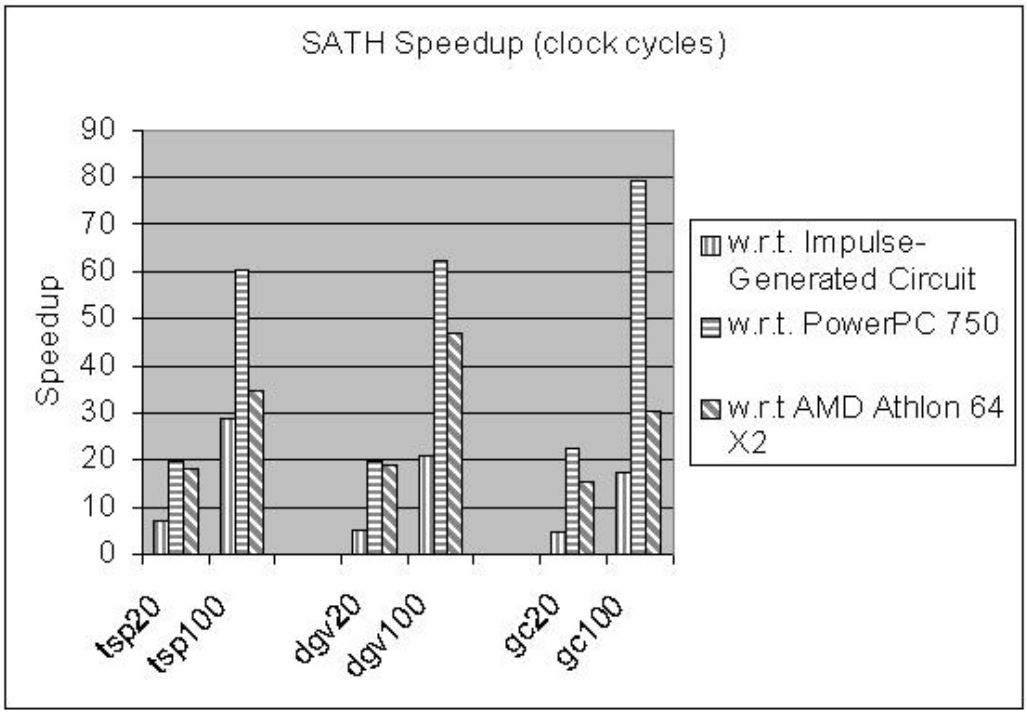


Fig. 5.9: Speedup of SATH-generated circuits comparing clock cycles per iteration with respect to other implementations.

(Fig. 5.10), flip-flops (Fig. 5.11), DSP48s (Fig. 5.12), BRAMs (Fig. 5.13), and RALFs (Fig. 5.14), respectively. On average, it appears that SATH-generated architectures use about half the resources of Impulse-generated architectures when the RALF unit of area measurement is employed. Additionally, in all cases but BRAMs, SATH-generated circuits are generally equally or more efficient on a resource-by-resource basis.

Because these designs are targeted for in-space applications, protection against radiation is necessary. Xilinx provides a plug-in tool called X-TMR [76] for automatically deriving a TMR version (triple modular redundancy) on the target device. This provides adequate fault tolerance against SEUs (single event upsets) at the expense of increasing the size of the circuit by 210 to 220

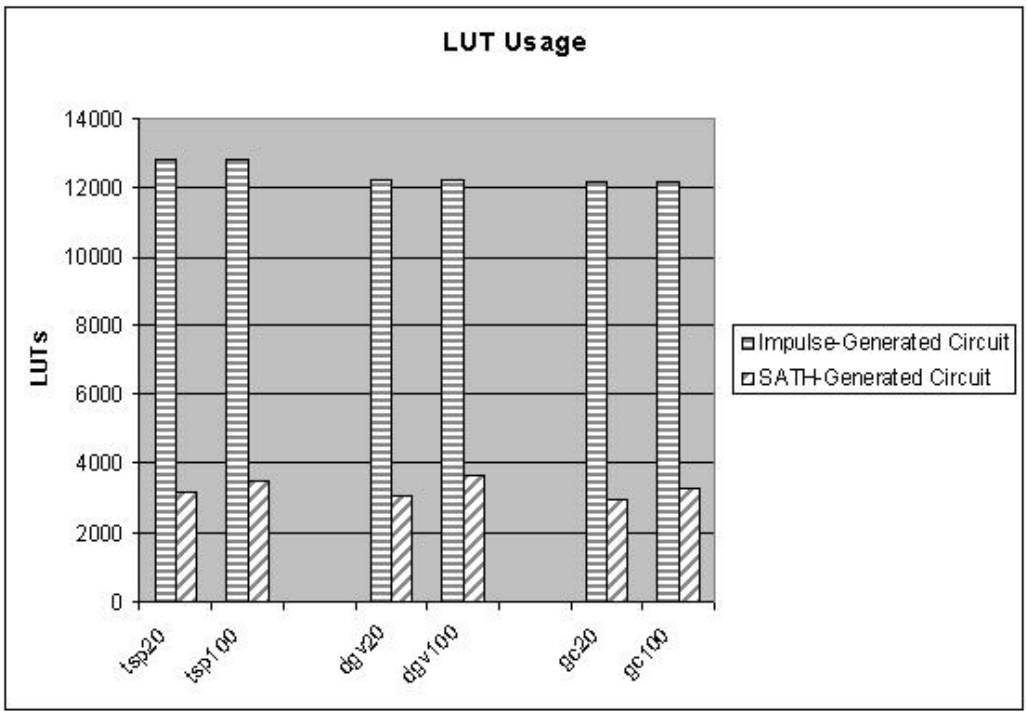


Fig. 5.10: Comparative LUT usage of circuits generated using Impulse and SATH.

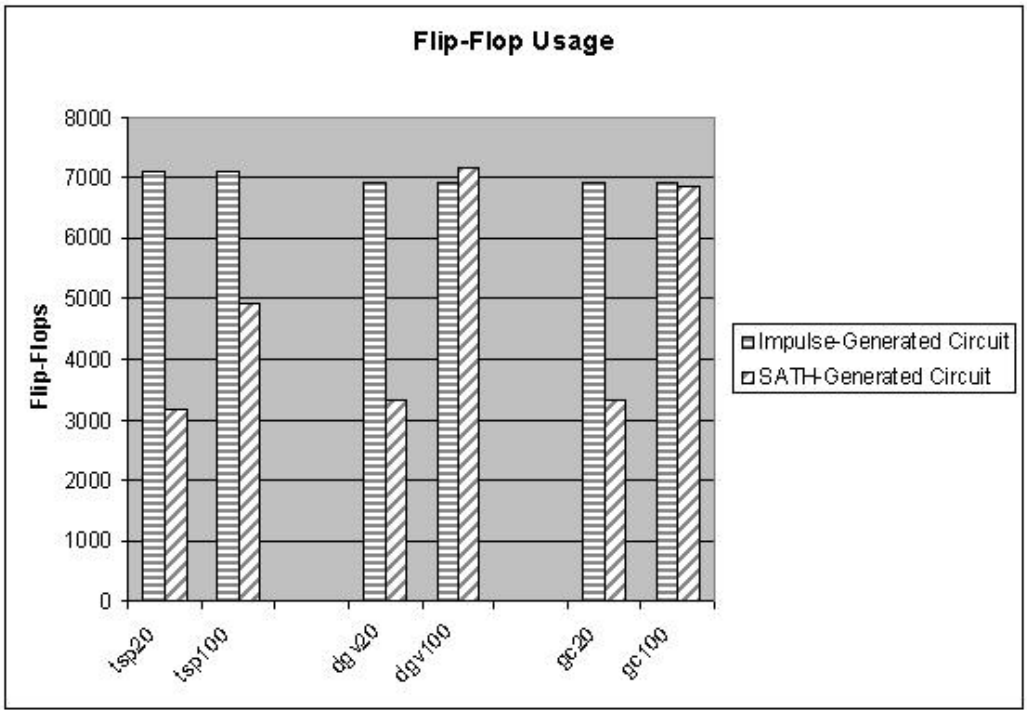


Fig. 5.11: Comparative flip-flop usage of circuits generated using Impulse and SATH.

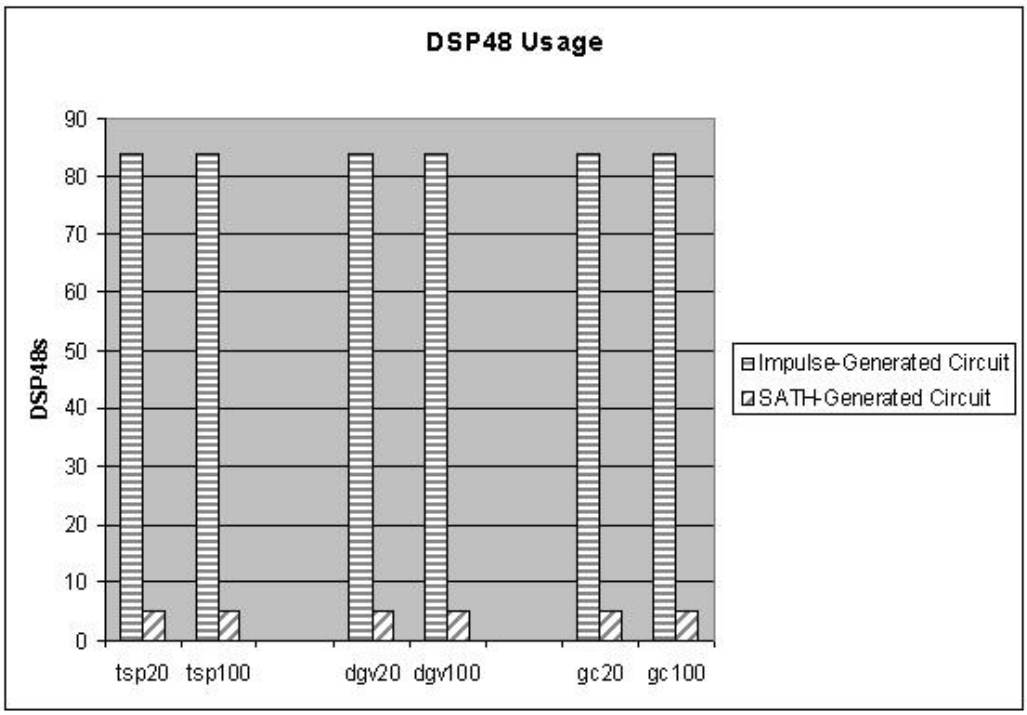


Fig. 5.12: Comparative DSP48 usage of circuits generated using Impulse and SATH.

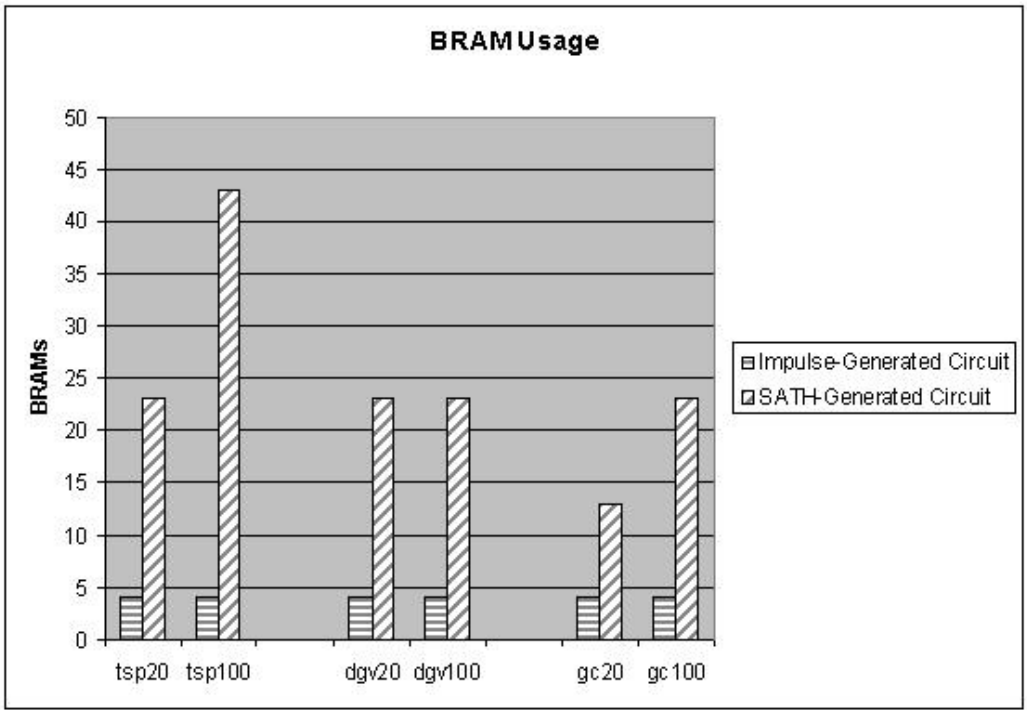


Fig. 5.13: Comparative BRAM usage of circuits generated using Impulse and SATH.

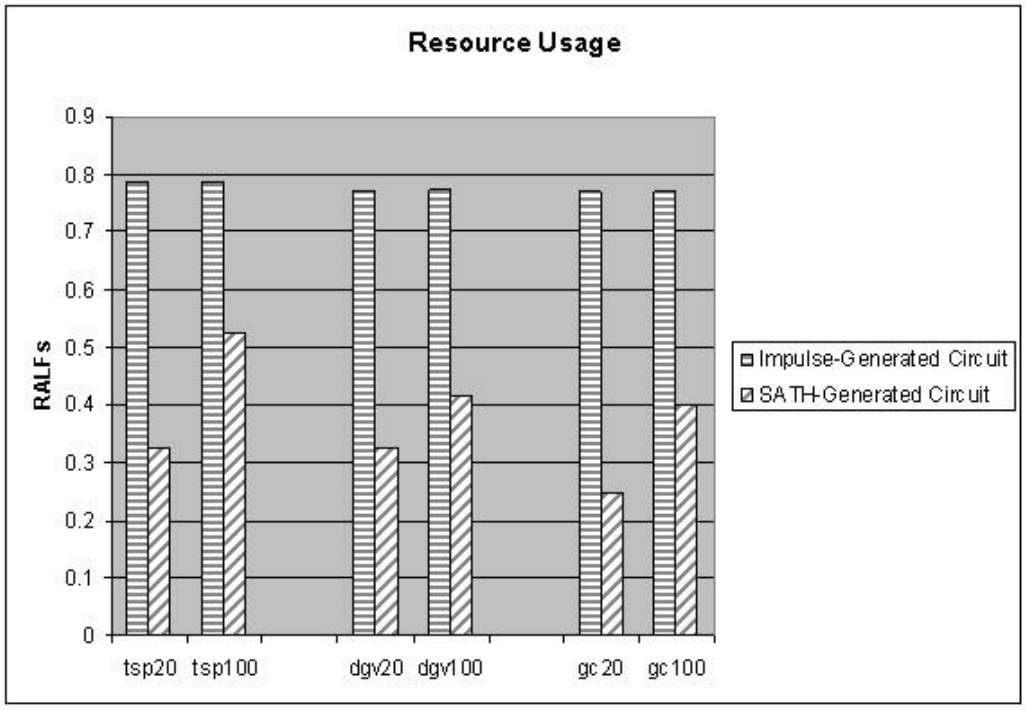


Fig. 5.14: Comparative RALF usage of circuits generated using Impulse and SATH.

Chapter 6

Conclusions and Future Work

In this paper, a novel methodology titled SATH (Simulated Annealing to Hardware) for generating modular hardware from application-specific source code has been presented. Performance of this algorithm for accelerating simulated annealing algorithms shows promising results. A method for the conversion of software source code into appropriate hardware accelerator circuits using a SAM algorithm described. The importance of selecting a proper architecture template is discussed. A method for compile-time approximation of required hardware resources is presented. Simulated annealing circuits generated using the proposed methodology are compared with the performance of commercially-available software-to-hardware conversion tools, with significant speedups attained across all test cases. While this research is specific to deriving hardware to accelerate simulated annealing codes, the techniques presented are applicable to a broad range of problem domains.

As discussed in chapter 2, various techniques exist for performing heuristic searches. Simulated annealing has been covered in this research. Additional work can be done to identify hardware templates and execution strategies for other methods, such as genetic algorithms or stochastic beam search.

Several optimization strategies from the field of compilers, as presented in [84] and [85], should also be explored. Most compiler optimization strategies are performed with knowledge of the specifics of the target architecture. Optimization techniques such as loop optimizations, code motion, strength reduction, and exploitation of mathematical properties such as the commutative and associative properties could play a role.

In the future, the technique of accelerator design using a high level template coupled with the described SAM method can be applied to other application domains. Additionally, work can be done to automate the process of identifying the form of the high level template.

References

[1] H. Emam, M. A. Ashour, H. Fekry, and A. M. Wahdan, "Introducing an fpga based genetic algorithms in the applications of blind signals separation," in *Proceedings of the 3rd IEEE International Workshop onSystem-on-Chip for Real-Time Applications*, pp. 123–127, 2003.

[2] A. Winterholler, M. Roman, D. Miller, J. Krause, and T. Hunt, "Automated core sample handling for future mars drill missions," in *8th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, Germany, 2005.

[3] NASA, "New space communications capabilities available for nasa's discovery and new frontier programs," 2006.

[4] J. Ramos, J. Samson, D. Lupia, I. Troxel, R. Subramaniyan, A. Jacobs, J. Greco, G. Cieslewski, J. Curreri, M. Fischer, E. Grobelny, A. George, V. Aggarwal, M. Patel, and R. Some, "High-performance, dependable multiprocessor," in *Aerospace Conference, IEEE*, J. Samson, Ed., 2006.

[5] W. Markiewicz, D. Titov, N. Ignatiev, H. Keller, D. Crisp, S. Limaye, R. Jaumann, R. Moissl, N. Thomas, L. Esposito, S. Watanabe, B. Fiethe, T. Behnke, I. Szemerey, H. Michalik, H. Perplies, M. Wedemeier, I. Sebastian, W. Boogaerts, S. Hviid, C. Dierker, B. Osterloh, W. Bker, M. Koch, H. Michaelis, D. Belyaev, A. Dannenberg, M. Tschimmel, P. Russo, T. Roatsch, and K. Matz, "Venus monitoring camera for venus express," *Planetary and Space Science*, vol. 55, pp. 1701–1711, 2007.

[6] B. Fiethe, H. Michalik, C. Dierker, B. Osterloh, and G. Zhou, "Reconfigurable system-on-chip data processing units for space imaging instruments," in *Design, Automation and Test in Europe*, pp. 1–6, 2007.

[7] S. Knight, G. Rabideau, S. Chien, B. Engelhardt, and R. Sherwood, "Casper: space exploration through continuous planning," *IEEE Intelligent Systems*, vol. 16, no. 5, pp. 70–75, 2001, 1541-1672.

[8] R. Sherwood, S. Chien, D. Tran, B. Cichy, R. Castano, A. Davies, and G. Rabideau, "The eo-1 autonomous sciencecraft," in *Small Satellite Conference*, Logan, UT, 2007.

[9] R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, and A. Fukunaga, "Using aspen to automate eo-1 activity planning," in *Proceedings of the IEEE Aerospace Conference*, vol. 3, pp. 145–152 vol.3, 1998.

[10] Xilinx, *Virtex-4 FPGA User Guide*, 2008.

[11] Xilinx, *Virtex-5 FPGA User Guide*, 2008.

[12] Altera, *Stratix IV Device Handbook*, 2008.

[13] W. Wolf, *FPGA-Based System Design*. Prentice Hall, 2004.

[14] S. A. Edwards, "The challenges of synthesizing hardware from c-like languages," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006, 0740-7475.

[15] C. E. Stroud, R. R. Munoz, and D. A. Pierce, "Behavioral model synthesis with cones," *IEEE Transactions on Design and Test of Computers*, vol. 5, no. 3, pp. 22–30, 1988, 0740-7475.

[16] D. Ku and G. De Micheli, "Hardwarec - a language for hardware design," *CSL-TR-90-419*, 1990.

[17] D. Galloway, "The transmogrifier c hardware description language and compiler for fpgas," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 136–144, 1995.

[18] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer, 2002.

[19] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A programming environment for the design of complex high speed asics," in *Proceedings of the Design Automation Conference*, pp. 315–320, 1998.

[20] R. J. Lipton, D. N. Serpanos, and W. H. Wolf, "Pdl++: an optimizing generator language for register transfer design," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 1135–1138 vol.2, 1990.

[21] D. Soderman and Y. Panchul, "Implementing c algorithms in reconfigurable hardware using c2verilog," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 339–342, 1998.

[22] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, cyber," in *Proceedings on the Design, Automation and Test in Europe Conference and Exhibition*, pp. 390–393, 1999.

[23] Celoxica, *Handel-C Reference Manual*, 2003.

[24] C. Hoare, "Communicating sequential processes," *CACM*, vol. 21, pp. 666–667, 1978.

[25] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura, "A c-based synthesis system, bach, and its application," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 151–155, 2001.

[26] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Kluwer, 2000.

[27] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale, "Trident: an fpga compiler framework for floating-point algorithms," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 317–322, 2005.

[28] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the 16th International Conference onVLSI Design*, pp. 461–466, 2003.

[29] M. Budiu and S. Goldstein, "Compiling application-specific hardware," in *Proc. FPL, LNCS*, pp. 853–863, Montpellier, France, 2002.

[30] A. Antola, M. D. Santambrogio, M. Fracassi, P. Gotti, and C. Sandionigi, "A novel hardware/software codesign methodology based on dynamic reconfiguration with impulse c and codeveloper," in *3rd Southern Conference on Programmable Logic*, pp. 221–224, 2007.

[31] P. Messmer, V. Ranjbar, D. Wade-Stein, and P. Schoessow, "Advanced accelerator control and instrumentation modules based on fpga," in *Particle Accelerator Conference*, pp. 506–508, 2007.

[32] G. Mehta, R. R. Hoare, J. Stander, and A. K. Jones, "Design space exploration for low-power reconfigurable fabrics," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, p. 4 pp., 2006.

[33] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partioning in hardware-software codesign," *Design Automation for Embedded Systems*, vol. 6, pp. 425–449, 2002.

[34] B. Miramond and J. M. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 366–371 Vol. 1, 2005.

[35] G. Ascia, V. Catania, and M. Palesi, "Design space exploration methodologies for ip-based system-on-a-chip," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, vol. 2, pp. II–364–II–367 vol.2, 2002.

[36] V. Catania, G. Ascia, M. Palesi, D. Patti, and A. G. Di Nuovo, "Fuzzy decision making in embedded system design," in *Proceedings of the 4th international conference on hardware/software codesign and system synthesis*, pp. 223–228, 2006.

[37] E. M. d. Icaya, V. Rodellar, C. Gonzalez, V. Peinado, and V. Garcia, "Design space exploration for an adaptive noise cancellation algorithm," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs*, pp. 1–7, 2006.

[38] C. Talarico, E. Rodriguez-Marek, and K. Min-sung, "Multi-objective design space exploration methodologies for platform based socs," in *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, p. 7 pp., 2006.

[39] C. Silvano, G. Agosta, and G. Palermo, "Efficient architecture/compiler co-exploration using analytical models," *Design Automation for Embedded Systems*, vol. 11, pp. 1–23, 2007.

[40] K. Kurt, R. Kaushik, S. Nadathur, and J. Yujia, "An automated exploration framework for fpga-based soft multiprocessor systems," in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, pp. 273–278, 2005.

[41] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. Ozturan, and G. Diindar, "Optimizing instruction-set extensible processors under data bandwidth constraints," in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 1–6, 2007.

[42] B. So, M. Hall, and P. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *Conference on Programming Language Design and Implementation*, pp. 165–176, Germany, 2002.

[43] S. Bilavarn, G. Gogniat, J. L. Philippe, and L. Bossuet, "Design space pruning through early estimations of area/delay tradeoffs for fpga implementations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1950–1968, 2006, 0278-0070.

[44] C. Silvano, D. Sciuto, D. Bruschi, and G. Beltrame, "Decision-theoretic exploration of multiprocessor platforms," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pp. 205–210, 2006.

[45] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *Computers and Digital Techniques, IEE Proceedings-*, vol. 153, no. 3, pp. 173–180, 2006, 1350-2387.

[46] T. L. Adam, K. Chandy, and J. Dickson, "A comparison of list scheduling for parallel processing systems," *Communications of the ACM*, vol. 17, pp. 685–690, 1974.

[47] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *24th Conference on Design Automation*, pp. 195–202, 1987.

[48] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989, 0278-0070.

[49] M. Y. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, 1990, 1045-9219.

[50] M. Shang, S. Sun, and Q. Wang, "An efficient parallel scheduling algorithm of dependent task graphs," in *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 595–598, 2003.

[51] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling precedence graphs in systems with inter processor communication times," *SIAM Journal on Computing*, vol. 5, no. 8, pp. 879–886, 1994.

[52] K. Yu-Kwong and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996, 1045-9219.

[53] S. Govindarajan and R. Vemari, "Improving the schedule quality of static-list time-constrained scheduling," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, p. 749, 2000.

[54] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments," in *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 149–155, 2003.

[55] S. Govindarajan and R. Vemari, "Cone-based clustering heuristic for list-scheduling algorithms," in *Proceedings of European Design and Test Conference*, pp. 456–462, 1997.

[56] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop, "Scheduling of conditional process graphs for the synthesis of embedded systems," in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 132–138, 1998.

[57] R. J. Cloutier and D. E. Thomas, "The combination of scheduling, allocation, and mapping in a single algorithm," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 71–76, 1990.

[58] J. Holland, *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[59] S. Russel and P. Norvig, *Artificial Intelligence A Modern Approach*, 2nd ed. Pearson, 2003.

[60] M. S. Hamid and S. Marshall, "Fpga realisation of the genetic algorithm for the design of grey-scale soft morphological filters," in *Proceedings of the International Conference on Visual Information Engineering*, pp. 141–144, 2003.

[61] H. E. Mostafa, A. I. Khadragi, and Y. Y. Hanafi, "Hardware implementation of genetic algorithm on fpga," in *Proceedings of the Twenty-First National Radio Science Conference*, pp. C9–1–9, 2004.

[62] S. D. Scott, A. Samal, and S. Seth, "Hga: A hardware-based genetic algorithm," in *Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays*, pp. 53–59, 1995.

[63] T. Wallace and Y. Leslie, "Hardware implementation of genetic algorithms using fpga," in *Proceedings of the 47th Midwest Symposium on Circuits and Systems*, vol. 1, pp. I–549–52 vol.1, 2004.

[64] Z. Zhenhuan, D. Mulvaney, and V. Chouliaras, "Investigation of a new genetic algorithm designed for system-on-chip realization," in *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 2981–2987, 2006.

[65] S. Narayanan and C. Purdy, "Hardware implementation of genetic algorithm modules for intelligent systems," in *Proceedings of the 48th Midwest Symposium on Circuits and Systems*, pp. 1733–1736 Vol. 2, 2005.

[66] T. Tachibana, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito, "General architecture for hardware implementation of genetic algorithm," in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 291–292, 2006.

[67] M. A. Vega-Rodriguez, R. Gutierrez-Gil, J. M. Avila-Roman, J. M. Sanchez-Perez, and J. A. Gomez-Pulido, "Genetic algorithms using parallelism and fpgas: the tsp as case study," in *International Conference Workshops on Parallel Processing*, pp. 573–579, 2005.

[68] T. Anantharaman and R. Bisiani, "A hardware accelerator for speech recognition algorithms," in *Proceedings of the 13th annual international symposium on Computer architecture*, pp. 216–223, Tokyo, Japan, 1986.

[69] M. Wrighton and A. DeHon, "Hardware-assisted simulated annealing with application for fast fpga placement," in *International Symposium on Field-Programmable Gate Arrays*, pp. 33–42, 2003.

[70] T. Tascione, *Introduction to the Space Environment*, 2nd ed. Krieger, 1994.

[71] M. Berg, "Fault tolerance implementation within sram based fpga design based upon the increased level of single event upset susceptibility," in *Proceedings of the 12th IEEE International On-Line Testing Symposium*, pp. 89–91, 2006.

[72] L. Yanmei, L. Dongmei, and W. Zhihua, "A new approach to detect-mitigate-correct radiation-induced faults for sram-based fpgas in aerospace application," in *Proceedings of the IEEE National Aerospace and Electronics Conference*, pp. 588–594, 2000.

[73] C. A. Hulme, H. H. Loomis, A. A. Ross, and Y. Rong, "Configurable fault-tolerant processor (cftp) for spacecraft onboard processing," in *Proceedings of the IEEE Aerospace Conference*, vol. 4, pp. 2269–2276 Vol.4, 2004.

[74] G. L. Smith and L. de la Torre, "Techniques to enable fpga based reconfigurable fault tolerant space computing," in *Proceedings of the IEEE Aerospace Conference*, p. 11 pp., 2006.

[75] G. V. Larchev and J. D. Lohn, "Evolutionary based techniques for fault tolerant field programmable gate arrays," in *Proceedings of the Second IEEE International Conference on Space Mission Challenges for Information Technology*, p. 8 pp., 2006.

[76] Xilinx, *Xilinx TMRTool*, 2008.

[77] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," in *International Conference on Field Programmable Logic and Applications*, pp. 1–6, 2006.

[78] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, pp. 200–209, 1962.

[79] P. Bernardi, M. Reorda, L. Sterpone, and M. Violante, "On the evaluation of seu sensitiveness in sram-based fpgas," in *Proceedings of the 10th IEEE International On-Line Testing Symposium*, M. S. Reorda, Ed., pp. 115–120, 2004.

[80] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of virtex fpga tmr design methodology," in *6th European Conference on Radiation and Its Effects on Components and Systems*, C. Carmichael, Ed., pp. 275–282, 2001.

[81] L. Sterpone and M. Violante, "A new partial reconfiguration-based fault-injection system to evaluate seu effects in sram-based fpgas," *IEEE Transactions on Nuclear Science*, vol. 54, no. 4, pp. 965–970, 2007, 0018-9499.

[82] D. Rea, D. Bayles, P. Kapcio, S. Doyle, and D. Stanley, "Powerpc rad750 - a microprocessor for now and the future," in *IEEE Aerospace Conference*, pp. 1–5, 2005.

[83] M. Garey and S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1st ed. W. H. Freeman and Company, 1978.

[84] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[85] Y. Srikant and P. Shankar, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, 2nd ed. CRC Press, 2008.

Appendices

Appendix A

Iterative Repair C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX_EVENTS 2000
#define INITIAL_TEMP 10000
#define COOLING_RATE 0.9999
#define STOP_THRESHOLD 0.0001

#define MAX_RESOURCE_TYPES 4
#define MAX_LATENCY 32
#define MAX_EDGES 99

void createInitialSchedule(int *current);
void anneal(int *current);
void copy(int *current, int *next);
void alter(int *next);
int evaluate(int *next);
void accept(int *current_val, int next_val, int *current, int *next,
   float temperature);
float adjustTemperature(float temperature);

int main()
{
   int current[MAX_EVENTS];

   createInitialSchedule(current);
   anneal(current);
   return 0;
}

void createInitialSchedule (int *current)
{
   int i;

   for (i=0; i<MAX_EVENTS; i++)
```

```
   {
      current[i] = 0;
   }
}

void anneal(int *current)
{
   float temperature;
   int current_val, next_val;
   int next[MAX_EVENTS];

   temperature = INITIAL_TEMP;
   current_val = RAND_MAX;
   while (temperature > STOP_THRESHOLD)
   {
      copy(current, next);
      alter(next);
      next_val = evaluate(next);
      accept(&current_val, next_val, current, next, temperature);
      temperature = adjustTemperature(temperature);
   }
}

void copy(int *current, int *next)
{
   int i;

   for (i=0; i<MAX_EVENTS; i++)
   {
      next[i] = current[i];
   }
}

void alter(int *next)
{
   int i, j;

   i = rand() % MAX_EVENTS;
   j = rand() % MAX_LATENCY;
   next[i] = j;
}

int evaluate (int *next)
{
   static int edge_source[MAX_EDGES] = {
       0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,
```

```
   21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
   41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,
   61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
   81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98};
static int edge_destination[MAX_EDGES] = {
   14,14,16,16,18,18,20,20,22,22,24,24,26,26,27,27,29,29,31,31,
   33,33,35,35,37,37,38,39,39,41,41,43,43,45,45,47,47,49,49,50,
   50,52,52,54,54,56,56,58,58,59,60,60,62,62,64,64,66,66,68,68,
   69,69,71,71,73,73,75,75,76,77,77,79,79,81,81,83,83,84,84,86,
   86,88,88,89,90,90,92,92,94,94,95,95,97,97,98,99,99,99,99};
static int resources[MAX_RESOURCE_TYPES] = {4, 4, 4, 4};
static int resource_usage[MAX_EVENTS] = {0,0,0,0,0,0,0,0,0,0,
   0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,
   3,3,3,3,3,3,3,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,
   2,3,3,3,3,3,3,3,3,0,0,0,0,0,0,1,1,1,1,1,2,2,2,2,3};

int i, j, start, stop, conflicts;
int t_matrix[MAX_LATENCY][MAX_RESOURCE_TYPES];

conflicts = 0;
start = MAX_LATENCY;
stop = 0;
for (i=0; i<MAX_RESOURCE_TYPES; i++)
{
   for (j=0; j<MAX_LATENCY; j++)
   {
      t_matrix[j][i] = 0;
   }
}
for (i=0; i<MAX_EVENTS; i++)
{
   if (next[i] < start)
   {
      start = next[i];
   }
   if (next[i] > stop)
   {
      stop = next[i];
   }
}
conflicts = stop - start;

for (i=0; i<MAX_EDGES; i++)
{
   if (next[edge_source[i]] >= next[edge_destination[i]])
   {
```

```
            conflicts += next[edge_source[i]] - next[edge_destination[i]] + 1;
         }
      }

      for (i=0; i<MAX_EVENTS; i++)
      {
         t_matrix[next[i]][resource_usage[i]]++;
      }
      for (i=0; i<MAX_LATENCY; i++)
      {
         for (j=0; j<MAX_RESOURCE_TYPES; j++)
         {
            if (t_matrix[i][j] > resources[j])
            {
               conflicts = conflicts + (t_matrix[i][j] - resources[j]);
            }
         }
      }
      return conflicts;
}

void accept(int *current_val, int next_val, int *current, int *next,
      float temperature)
{
      int delta_e, i;
      float p, q;

      delta_e = next_val - *current_val;
      if (delta_e <= 0)
      {
         for (i=0; i<MAX_EVENTS; i++)
         {
            current[i] = next[i];
         }
         *current_val = next_val;
      }
      else
      {
         p = exp(-((float)delta_e)/temperature);
         q = (float) rand() / (float) RAND_MAX;
         if (q < p)
         {
            for (i=0; i<MAX_EVENTS; i++)
            {
               current[i] = next[i];
            }
```

```
         *current_val = next_val;
      }
   }
}

float adjustTemperature(float temperature)
{
   return temperature * COOLING_RATE;
}
```

Appendix B

TSP C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define MAX_EVENTS 20
#define INITIAL_TEMPERATURE 10000
#define COOLING_RATE  0.9999
#define STOP_THRESHOLD 0.0001

void anneal(int *current);
void copy(int *current, int *next);
void alter(int *next);
int evaluate(int *next);
void accept(int *current_val, int next_val, int *current, int *next,
   float temperature);
float adjustTemperature();

int main()
{
   static int current[MAX_EVENTS] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15, 16, 17, 18, 19};

   srand(time(NULL));
   anneal(current);
   return 0;
}

void anneal(int *current)
{
   float temperature;
   int current_val, next_val;
   int next[MAX_EVENTS];

   temperature = INITIAL_TEMPERATURE;
   current_val = RAND_MAX;
```

```
   while (temperature > STOP_THRESHOLD)
   {
      copy(current, next);
      alter(next);
      next_val = evaluate(next);
      accept(&current_val, next_val, current, next, temperature);
      temperature = adjustTemperature();
   }
}

void copy(int *current, int *next)
{
   int i;

   for (i=0; i<MAX_EVENTS; i++)
   {
      next[i] = current[i];
   }
}

void alter(int *next)
{
   int a, b, temp;

   a = rand() % MAX_EVENTS;
   b = rand() % MAX_EVENTS;
   temp = next[a];
   next[a] = next[b];
   next[b] = temp;
}

int evaluate (int *next)
{
   const int x_pos[MAX_EVENTS] = {27, 32, 91, 60, 36, 64, 32, 9, 7,
      64, 2, 28, 41, 4, 38, 33, 79, 65, 45, 57};
   const int y_pos[MAX_EVENTS] = {20, 17, 98, 83, 35, 77, 41, 61,
      0, 55, 17, 70, 4, 92, 25, 59, 16, 66, 39, 73};

   int distance, i;

   distance = 0;
   for (i=0; i<MAX_EVENTS-1; i++)
   {
      distance += abs(x_pos[next[i]] - x_pos[next[i+1]]) +
         abs(y_pos[next[i]] - y_pos[next[i+1]]);
   }
```

```
   return distance;
}

void accept(int *current_val, int next_val, int *current, int *next,
   float temperature)
{
   int delta_e, i;
   float p, r;

   delta_e = next_val - *current_val;
   if (delta_e <= 0)
   {
      for (i=0; i<MAX_EVENTS; i++)
      {
         current[i] = next[i];
      }
      *current_val = next_val;
   }
   else
   {
      p = exp(-((float)delta_e)/temperature);
      r = (float) rand() / (float) RAND_MAX;
      if (r < p)
      {
         for (i=0; i<MAX_EVENTS; i++)
         {
            current[i] = next[i];
         }
         *current_val = next_val;
      }
   }
}

float adjustTemperature()
{
   static float temperature = INITIAL_TEMPERATURE;

   temperature = temperature * COOLING_RATE;
   return temperature;
}
```

Appendix C

Graph Coloring C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define MAX_EVENTS 20
#define INITIAL_TEMPERATURE 10000
#define COOLING_RATE  0.9999
#define STOP_THRESHOLD 0.0001
#define MAX_EDGES 30
#define MAX_COLORS 3

void anneal(int *current);
void copy(int *current, int *next);
void alter(int *next);
int evaluate(int *next);
void accept(int *current_val, int next_val, int *current, int *next,
   float temperature);
float adjustTemperature();

int main()
{
   static int current[MAX_EVENTS] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15, 16, 17, 18, 19};

   srand(time(NULL));
   anneal(current);
   return 0;
}

void anneal(int *current)
{
   float temperature;
   int current_val, next_val;
   int next[MAX_EVENTS];
```

```
   temperature = INITIAL_TEMPERATURE;
   current_val = RAND_MAX;
   while (temperature > STOP_THRESHOLD)
   {
      copy(current, next);
      alter(next);
      next_val = evaluate(next);
      accept(&current_val, next_val, current, next, temperature);
      temperature = adjustTemperature();
   }
}

void copy(int *current, int *next)
{
   int i;

   for (i=0; i<MAX_EVENTS; i++)
   {
      next[i] = current[i];
   }
}

void alter(int *next)
{
   int i, j;

   i = rand() % MAX_EVENTS;
   j = rand() % MAX_COLORS;
   next[i] = j;
}

int evaluate (int *next)
{
   const int a[MAX_EDGES] = {15, 9, 14, 12, 15, 19, 3, 3, 3, 18, 1,
      10, 3, 17, 4, 19, 17, 7, 16, 13, 15, 2, 12, 9, 1, 0, 14, 8, 14, 15};
   const int b[MAX_EDGES] = {3, 19, 0, 11, 9, 15, 11, 7, 19, 16, 7,
      16, 2, 16, 6, 14, 6, 13, 4, 3, 12, 1, 18, 5, 18, 6, 7, 2, 15, 18};
   int i, violations;

   violations = 0;
   for (i=0; i<MAX_EDGES; i++)
   {
      if (next[a[i]] == next[b[i]])
      {
         violations++;
      }
```

```
   }
   return violations;
}

void accept(int *current_val, int next_val, int *current, int *next,
   float temperature)
{
   int delta_e, i;
   float p, r;

   delta_e = next_val - *current_val;
   if (delta_e <= 0)
   {
      for (i=0; i<MAX_EVENTS; i++)
      {
         current[i] = next[i];
      }
      *current_val = next_val;
   }
   else
   {
      p = exp(-((float)delta_e)/temperature);
      r = (float) rand() / (float) RAND_MAX;
      if (r < p)
      {
         for (i=0; i<MAX_EVENTS; i++)
         {
            current[i] = next[i];
         }
         *current_val = next_val;
      }
   }
}

float adjustTemperature()
{
   static float temperature = INITIAL_TEMPERATURE;

   temperature = temperature * COOLING_RATE;
   return temperature;
}
```

Appendix D

Dependency Graph Violation C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define MAX_EVENTS 20
#define INITIAL_TEMPERATURE 10000
#define COOLING_RATE  0.9999
#define STOP_THRESHOLD 0.0001
#define MAX_LATENCY 6
#define MAX_EDGES 30

void anneal(int *current);
void copy(int *current, int *next);
void alter(int *next);
int evaluate(int *next);
void accept(int *current_val, int next_val, int *current, int *next,
   float temperature);
float adjustTemperature();

int main()
{
   static int current[MAX_EVENTS] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
      11, 12, 13, 14, 15, 16, 17, 18, 19};

   srand(time(NULL));
   anneal(current);
   return 0;
}

void anneal(int *current)
{
   float temperature;
   int current_val, next_val;
   int next[MAX_EVENTS];
```

```
   temperature = INITIAL_TEMPERATURE;
   current_val = RAND_MAX;
   while (temperature > STOP_THRESHOLD)
   {
      copy(current, next);
      alter(next);
      next_val = evaluate(next);
      accept(&current_val, next_val, current, next, temperature);
      temperature = adjustTemperature();
   }
}

void copy(int *current, int *next)
{
   int i;

   for (i=0; i<MAX_EVENTS; i++)
   {
      next[i] = current[i];
   }
}

void alter(int *next)
{
  int i, j;

  i = rand() % MAX_EVENTS;
  j = rand() % MAX_LATENCY;
  next[i] = j;
}


int evaluate (int *next)
{
   const int source[MAX_EDGES] = { 14, 17, 7, 1, 10, 9, 13, 8, 14,
      9, 18, 15, 9, 6, 2, 11, 7, 6, 7, 6, 7, 15, 0, 5, 13, 3, 4, 0, 5, 5};
   const int dest[MAX_EDGES] = {18, 19, 17, 9, 16, 15, 16, 14, 17,
      11, 19, 19, 17, 8, 7, 16, 14, 19, 16, 12, 13, 16, 12, 10, 14, 12,
      7, 19, 17, 15};
   int i, conflicts;

   conflicts = 0;
   for (i=0; i<MAX_EDGES; i++)
   {
      if (next[source[i]] >= next[dest[i]])
      {
```

```
         conflicts = conflicts + (next[source[i]] - next[dest[i]]);
      }
   }
   return conflicts;
}

void accept(int *current_val, int next_val, int *current, int *next,
   float temperature)
{
   int delta_e, i;
   float p, r;

   delta_e = next_val - *current_val;
   if (delta_e <= 0)
   {
      for (i=0; i<MAX_EVENTS; i++)
      {
         current[i] = next[i];
      }
      *current_val = next_val;
   }
   else
   {
      p = exp(-((float)delta_e)/temperature);
      r = (float) rand() / (float) RAND_MAX;
      if (r < p)
      {
         for (i=0; i<MAX_EVENTS; i++)
         {
            current[i] = next[i];
         }
         *current_val = next_val;
      }
   }
}

float adjustTemperature()
{
   static float temperature = INITIAL_TEMPERATURE;

   temperature = temperature * COOLING_RATE;
   return temperature;
}
```